PARALLELIZED KD TREE FOR PARALLEL K-NEAREST NEIGHBORS SEARCH

NEELANSH KAABRA DOREEN VALMYR

CONTENTS

LIST OF FIGURES

Computer Science Department, Carnegie Mellon University, Pittsburgh, USA

1 SUMMARY

Our project aims to create a robust parallel kNN search algorithm by leveraging a parallelized, lock-free KD tree for efficient data traversal. The parallel KD tree will be utilized to parallelize the search process, distributing the workload across multiple threads. The overarching goal is to achieve high-performance kNN search in large datasets within a parallel computing environment, specifilly the GHC and Bridges-2 machines. This approach is designed to significantly reduce search times and improve efficiency in handling large-scale data. Our 128 core implementation on Bridges-2 achieves a 70x speedup when searching for 1 thousand neighbors in a dataset of 2 million points.

2 BACKGROUND

2.1 KD Tree

2.1.1 Defining KD Tree

A KD-tree (k-dimensional tree) is a space-partitioning data structure used for organizing points in a k-dimensional space. It's particularly useful for applications requiring efficient searches, like nearest neighbor queries in multidimensional data. KD-trees recursively divide the space into two half-spaces at each node, using a median value of the points to determine the splitting plane. This structure enables efficient search operations by reducing the number of comparisons needed to find a point or a group of points within a given range.

The project aims to accelerate k-nearest neighbors (kNN) search, crucial for spatial databases, machine learning, and data mining. Its primary objective is to improve efficiency in identifying the k-nearest neighbors in large datasets. This is achieved by utilizing a parallelized KD tree, enhancing data organization and access in multidimensional spaces. The focus is on making the KD tree lock-free, optimizing the search process, and ensuring faster, more efficient computation in handling extensive and complex data queries.

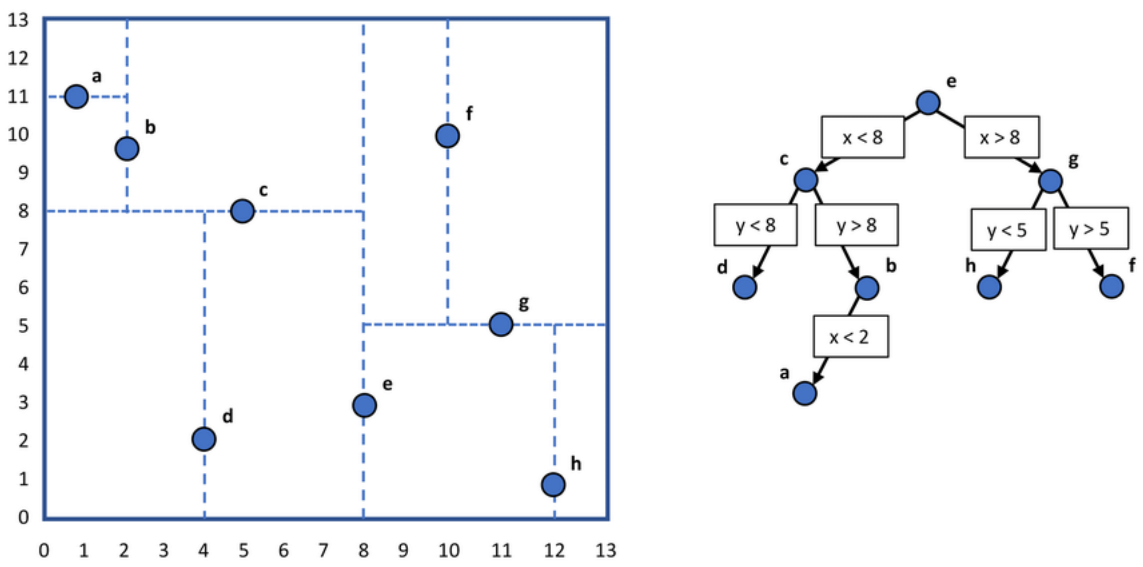


Figure 1: A 2 dimensional KD Tree

```
Function buildKDTreeImplSeq(data, depth, k):
    if data is empty:
        return null

    axis = depth % k
    median = size of data / 2
    nth_element(data, median, comparison function based on axis)

    Create a new KDNode node
    node.features = features of the median data point
    node.label = label of the median data point

    leftData = data from start to median
    node.left = recursively call buildKDTreeImplSeq with leftData, depth + 1, k

    rightData = data from median + 1 to end
    node.right = recursively call buildKDTreeImplSeq with rightData, depth + 1, k

    return node

Function buildKDTreeSeq(data, depth, k):
    root = call buildKDTreeImplSeq with data, depth, k
    dimensions = k
```

2.1.2 *Data Structures and Operations*

We created our own *KDNode* class containing a vector of features, a label, and two pointers to the node's left and right children. This design enables the formation of a tree-like structure where nodes are organized based on their features.

The program receives two inputs: the number of features associated with the points in the dataset and the selected dataset itself. The algorithm produces the constructed KD Tree as output, along with the corresponding time taken for tree construction.

The principal operations conducted on this class include building and printing the KD Tree. After parsing the data points, the vector of points is directed to the *buildKDTree* function for tree construction. The data points are organized into a *Datapoint* vector, containing the list of features, the label, and the threadId of each point—utilized for the parallel implementation using OpenMP.

Concerning the computationally intensive aspect of KD Tree construction, it entails traversing through all data points and inserting them in the correct locations, involving several recursive calls for each point. To enhance efficiency, we parallelized this phase of the implementation by assigning different tasks to these recursive calls.

Dependencies are inherent in the recursive construction process, as the decision to traverse left or right depends on the features of the data points and the splitting dimension of the current node. However, these dependencies are localized to each specific node in the tree. The construction process exhibits data parallelism, allowing concurrent processing of distinct nodes, and the tree structure promotes locality, enabling efficient cache utilization during traversal.

2.2 KNN

2.2.1 *Defining kNN*

K-Nearest Neighbors (kNN) is a supervised machine learning algorithm used for classification and regression. The kNN search involves finding the k nearest neighbors of a given query point in a dataset based on a distance metric, commonly **Euclidean distance**. The common steps for this algorithm are as follows:

1. Iterate over all points and calculate the distance from the current point to the target point
2. Create a list of the k nearest neighbors based on the distance
3. Find the most common label from the k nearest neighbors and assign that label to the target point

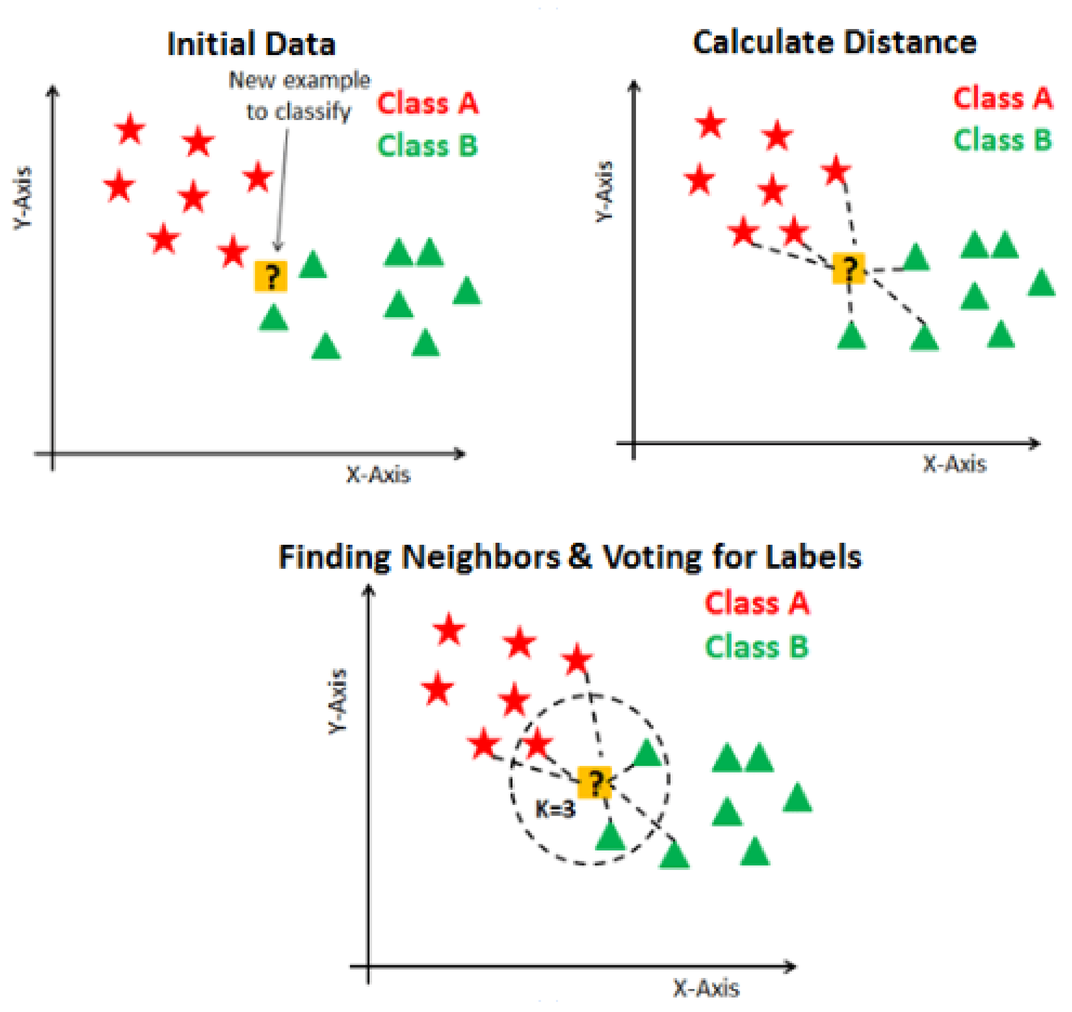


Figure 2: kNN Algorithm

The hierarchical structure of a KD Tree, however, enables a more efficient search by eliminating large portions of the data space early in the process. The steps for completing KNN search using a KD Tree are as follows:

1. Start from the root
2. Traverse the tree to the section where the new point belongs based on feature currently being analyzed
3. Add the node to the queue in the correct/sorted location

   a) If there are less than k neighbors in the queue, add the node to the queue
   b) If the number of nodes in the queue is equal to *k*, check if the node can be placed in the queue using the distance as the priority
4. Check to see if there are "better" nodes on the other side
5. After creating the list of *k* nearest neighbors, determine the target point's label based on the most common label of the neighbors

2.2.2 Data Structures and Operations

We established a *KNN* class that encapsulates shared functionalities and operations required by both the parallel and sequential versions of the program. This class encompasses various search functions, such as *kNNSearchParallelOpenMP*, *kNNSearchParallelMPI*, and *kNNSearch*, each designed for specific purposes. Complementary to these search functions, which retrieve nearest neighbors as implied by their names, are additional functions for printing the identified nearest neighbors (*printNearestNeighbors*) and determining the target point's label based on its nearest neighbors (*findTargetLabel*).

The KNN program takes several inputs, including the target point, the number of features in the target point, the chosen dataset, and the specified number of neighbors for consideration. In addition to the data structures employed for the KD Tree, an extra structure called *DistanceNode* was introduced. This structure facilitates the straightforward tracking of distances from a particular node in the KD Tree to the target point.

The most computationally demanding aspect of this program centered around determining the *k* nearest neighbors. This process entailed navigating through the KD Tree and assembling a list of nearest neighbors through distance calculations. Additionally, the procedure involved organizing the nearest neighbor list to adopt a priority queue format. Given the substantial computational load required by the algorithm, we optimized it for parallel execution, primarily leveraging MPI. This approach enabled distinct cores to construct smaller KD Trees based on their individual sets of data points, allowing them to search independently without interference from other cores.

A dependency exists in node traversal, where the choice to move left or right depends on the features of the target point and the current node's splitting dimension. However, this dependency is confined to each individual node. The processing of nodes within the KD Tree demonstrates data parallelism, allowing separate threads or processes to operate on distinct nodes concurrently without significant data dependencies. Nodes in the KD Tree display locality, as traversal typically involves accessing neighboring nodes in the tree structure. This characteristic promotes efficient cache utilization and enhanced memory locality.

3 APPROACH

The parallel computing frameworks and methods used in this project include **OpenMP**, **Open MPI**, and **CAS**.

The KD Tree was parallelized using OpenMP and lock free insertions into the KD Tree were made using compare-and-swap (CAS) operations.

The kNN search algorithm was parallelized using both OpenMP and MPI.

3.1 Parallelization Strategies

KD TREE PARALLELIZATION The construction of the KD tree is parallelized to expedite the process of organizing the dataset. Each thread is responsible for constructing specific portions of the tree. Compare-and-Swap is employed to manage concurrent access to shared data structures during the construction of the tree.

The pseudocode below implements a parallel KD-tree construction algorithm using OpenMP, a parallel programming model in **C++**. The pragma directives are left out for simplicity, but the parallel sections are explicitly noted.

Parallelization is introduced in the early stages of the KD-tree construction
(depth < MAX_PARALLEL_DEPTH). OpenMP pragmas (#pragma omp parallel, #pragma omp single, and #pragma omp task) are used to create parallel tasks for building left and right subtrees. Parallelism is limited to a predefined tree depth (MAX_PARALLEL_DEPTH), beyond which the algorithm reverts to sequential processing, thus balancing the computational efficiency of parallel processing with the overhead of managing parallel tasks.

OpenMP simplifies the parallelization of code with directives and handles thread creation and management allowing for efficient utilization of multi-core processors, reducing the overall runtime for KD-tree construction. While parallelism speeds up construction for large datasets, it introduces complexity due to the need for thread synchronization and potential data race conditions and therefore the choice of MAX_PARALLEL_DEPTH impacts performance and needs to be optimized based on the hardware and dataset size.

The parallel approach significantly improves the performance of the KD-tree construction, especially beneficial for large, high-dimensional datasets. It achieves a balance between the computational overhead of parallel processing and the efficiency gains from reduced execution time.

```
Function buildKDTreeImplPar(data, depth, k):
    if data is empty:
        return null

    axis = depth % k
    median = size of data / 2
    nth_element(data, median, based on axis)

    Create a new KDNode node
    node.features = features of median data point
    node.label = label of median data point

    if depth < MAX_PARALLEL_DEPTH:
        parallel section:
            task 1:
                leftData = data from start to median
                node.left = recursively call buildKDTreeImplPar with
                leftData, depth + 1, k
            task 2:
                rightData = data from median + 1 to end
                node.right = recursively call buildKDTreeImplPar with
                rightData, depth + 1, k
        wait for tasks to complete
    else:
        leftData = data from start to median
        node.left = buildKDTreeImplPar(leftData, depth + 1, k)
        rightData = data from median + 1 to end
        node.right = buildKDTreeImplPar(rightData, depth + 1, k)

    return node
```

KNN PARALLELIZATION: MPI The core steps of the algorithm involve distributing the dataset among multiple MPI processes, constructing local KDTrees on each process, performing a KNN search on the local KDTree, and then using MPI communication to gather and combine the results on the root process.

To utilize MPI effectively, programmers typically need to make changes to their existing sequential code or design new algorithms that can take advantage of parallelism.

In the case of the kNN algorithm, we defined explicitly what data each process operated on. This involved using block assignment to partition the points parsed from a dataset. If the number of points was not divisible by the number of processes, the last process would operate on the remaining points. As explained in the Results section, this may have been a cause of load imbalance. Each process would then build a separate KD Tree using its own data, and using this KD Tree, the process would determine the target point's *k* nearest neighbors. In the end, the first process would gather and combine these intermediate results, keeping on the *k* neighbors.

```
function kNNSearchParallelMPI(data, target, k, rank, size):
    Each process receives a portion of the data
    localData = distributeData(rank, size, data)

    Build a local KDTree using the local data
    localKDTree = buildKDTree(localData)

    Perform KNN search on the local KDTree
    localNearestNeighbors = kNNSearch(localKDTree, target, k)

    Gather the local nearest neighbors from all processes
    allNearestNeighbors = gatherNearestNeighbors(localNearestNeighbors, size)

    On process rank 0, combine and sort the results
    if rank == 0:
        sortedNearestNeighbors = combineAndSort(allNearestNeighbors)

        Select the top k nearest neighbors
        result = sortedNearestNeighbors[:k]

    return result
```

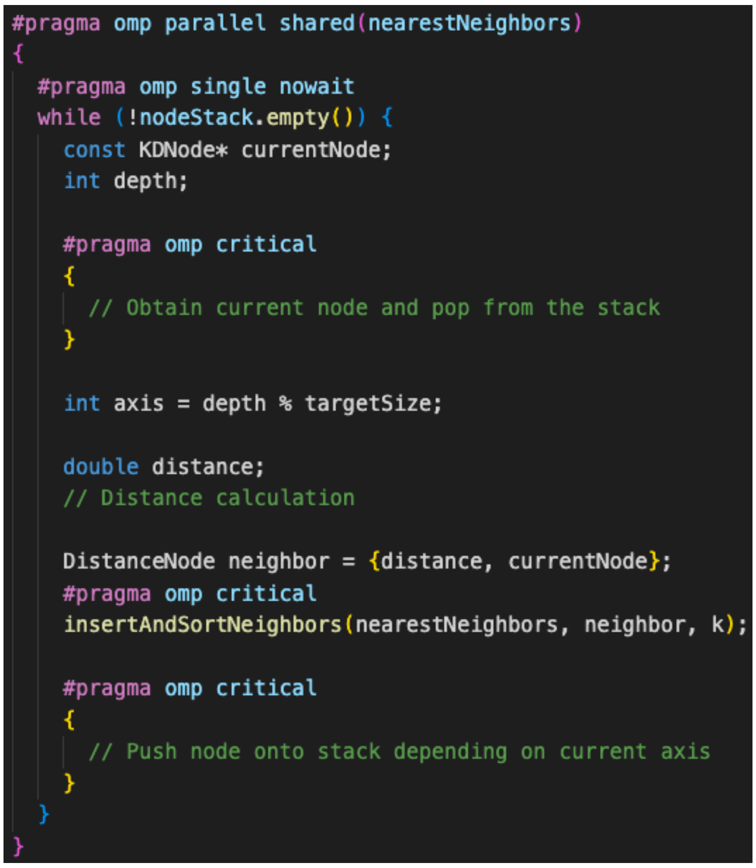
KNN PARALLELIZATION: OPENMP Using OpenMP for parallelizing an application is generally straightforward for many types of parallel tasks, especially those with regular and predictable patterns. However, it can become challenging for certain applications.

Parallelizing kNN search with OpenMP can be straightforward for certain aspects of the algorithm. For instance, the distance computations between points, a key step in KNN, can be parallelized easily by employing OpenMP to parallelize "for" loops iterating over data points. Additionally, the sorting of distances to find nearest neighbors is another task that can benefit from OpenMP, particularly when applied to parallelize sorting algorithms.

However, challenges arise due to data dependencies. The KNN algorithm inherently has dependencies between data points, as finding neighbors for one point may depend on the results for another. Managing these data dependencies in a parallel context requires careful synchronization to avoid race conditions and ensure accurate results.

For this project, using OpenMP to parallelize the kNN search proved to be a challenge. The datasets that we created had a maximum of 10 features per point. Since the number of features is so low, parallelizing the distance calculation would hinder the performance of the program as a result of overhead. The number of features could not be increased due to memory considerations when using CMU's gates machines. Therefore, the main way to parallelize was using the "parallel" directive when iterating over nodes in the KD Tree.

In the code below, OpenMP is utilized to parallelize the k-nearest neighbors (KNN) search algorithm. The main parallel region is initiated with #pragma omp parallel, enabling multiple threads to work concurrently. The #pragma omp single nowait directive ensures that certain tasks are executed by only one thread, such as initializing data structures. Critical sections, denoted by #pragma omp critical, safeguard shared data structures, including the nodeStack and nearestNeighbors vector, preventing simultaneous access by multiple threads to avoid conflicts. The search strategy is distributed among threads, allowing them to collectively process different parts of the dataset simultaneously. Overall, OpenMP is applied to enhance the efficiency of the KNN search by parallelizing key components and leveraging multiple threads for concurrent execution. However, for small datasets, the performance of the program will lessen when using OpenMP as a result of overhead from creating tasks.


```
#pragma omp parallel shared(nearestNeighbors)
{
  #pragma omp single nowait
  while (!nodeStack.empty()) {
    const KDNode* currentNode;
    int depth;

    #pragma omp critical
    {
      // Obtain current node and pop from the stack
    }

    int axis = depth % targetSize;

    double distance;
    // Distance calculation

    DistanceNode neighbor = {distance, currentNode};
    #pragma omp critical
    insertAndSortNeighbors(nearestNeighbors, neighbor, k);

    #pragma omp critical
    {
      // Push node onto stack depending on current axis
    }
  }
}
```

Figure 3: Code snippet to display use of OpenMP in kNN algorithm

LOCK FREE KDTREE: CAS The provided code demonstrates a lock-free KD-tree construction in C++ using atomic operations, fundamental to lock-free data structures. Lock-free structures allow concurrent access and modification by multiple threads without traditional lock-based synchronization, thus reducing the overhead and potential bottlenecks caused by locks. In this KD-tree implementation, atomic pointers ('std::atomic<KDNode*>') ensure thread-safe node insertion and updates. The recursive function 'insertRecursiveLockFree' handles node insertions atomically and directs the node placement in the tree based on dimensional comparisons. This approach enhances concurrent performance, particularly in multi-threaded applications, by facilitating non-blocking operations on the data structure.

In the provided code, the "compare and exchange" operation is a key component of the lock-free approach. This operation, executed via 'compare_exchange_weak', is used during the insertion of new nodes into the KD-tree. It atomically compares the current node pointer with a null pointer; if they match (indicating no concurrent update has happened), it safely inserts the new node. This operation is crucial for ensuring that multiple threads can modify the tree without conflicting with each other, enhancing the efficiency and safety of concurrent operations in a multi-threaded environment.

```
    Function insertLockFree(dataPoint, depth, k):
    Create a new KDNode node with dataPoint's features and label
    Call insertRecursiveLockFree on root with node, depth, and k

Function insertRecursiveLockFree(current, node, depth, k):
    If current node is null, attempt to insert the new node atomically (CAS)
    If insertion is successful, return

    Calculate dimension for comparison (dim = depth % k)
    If node's feature at dim is less than current's feature at dim:
        Recursively call insertRecursiveLockFree on current's left subtree
    Else:
        Recursively call insertRecursiveLockFree on current's right subtree
```

3.2 Parallel Machines

We utilized 2 types of machines to test the performance of our parallel implementations:

- GHC clusters
  - 8 cores
  - CPU: Intel i7-9700 (8) @ 4.700GHz
- Bridges-2 machines at the Pittsburgh Supercomputing Center (PSC)
  - 128 cores per node
  - 2.25-3.40GHz

The performance of the parallel KD Tree construction was tested on the GHC machines with different sizes of datasets.

The performance of the parallel kNN search was tested on the GHC machines, since the speedup only becomes notible in this algorithm for large datasets which can be partitioned with more nodes. The dataset ranged from 50 to 2 million points, each point having 6 to 10 features depending on the dataset.

3.3 Analysis and Review

Our exploration into parallel computing with the construction of a KD-tree using OpenMP and the parallelization of a kNN algorithm via MPI reveals the intricate dynamics between computational efficiency and the architectural constraints of parallel systems. The KD-tree, built using OpenMP, capitalized on the multi-threading capabilities to expedite data organization significantly. By assigning tree construction to different threads and employing CAS operations for lock-free insertions, we mitigated synchronization bottlenecks, achieving notable speedups on the GHC clusters equipped with Intel i7 processors and 8 cores.

However, the parallel kNN search presented a more complex challenge. We initially pursued two distinct paths: one using OpenMP and the other employing MPI. Our OpenMP implementation aimed to exploit thread-level parallelism; however, it faced challenges due to the overhead of thread synchronization and the complexity of managing shared data structures, especially with smaller datasets on the GHC machines. On the other hand, the MPI-based approach, which we ultimately favored for its superior performance, was better suited for the distributed memory environment of the Bridges-2 machines at the Pittsburgh Supercomputing Center, which had up to 128 cores per node.

The MPI strategy involved distributing the dataset across multiple processes, each with its local KD-tree. This method allowed for a scalable and efficient search process, as each MPI process could independently perform the kNN search on its subset of the data. The results from each process were then gathered and combined, a task that MPI handles well through its communication protocols. This distributed approach not only improved data locality but also reduced the communication and synchronization overhead that impeded the OpenMP implementation.

Our iterative process of development and optimization revealed that while OpenMP offered some benefits, MPI was better suited for large-scale data due to its ability to handle distributed data and computation more effectively. The MPI approach provided a clear pathway to scaling up the kNN search, resulting in substantial speedups, particularly as the dataset size increased. This decision was reinforced by the performance plateau observed with OpenMP as we scaled up the number of threads, which was not as pronounced with MPI, signifying its capability to manage larger datasets and more complex searches more efficiently.

Ultimately, the choice of MPI over OpenMP for the kNN search underscores a critical lesson in parallel computing: the importance of matching the parallelization strategy to the specific requirements of the dataset and the computational environment. Our journey to this solution was marked by rigorous testing, analysis, and a willingness to pivot our strategy based on performance data.

With larger datasets, such as those containing 2 million points and searches for up to 100,000 neighbors, we encountered the non-linearities of speedup, emphasizing the law of diminishing returns in a high-process-count regime. This effect was attributed to the intricate interplay of load balancing, algorithmic scalability, and the overhead from increased process management. Our data structures and operations, while ideally mapped onto the cores and threads to exploit data parallelism, were not immune to the physical limitations of the hardware, such as cache sizes and memory transfer rates.

The journey to our final solution was iterative and filled by trials that tested various optimization strategies. Some attempts did not yield the expected improvements, often due to the overheads outweighing the benefits of additional parallelism. In these cases, we had to carefully adjust our approach, such as refining the MAX_PARALLEL_DEPTH to balance workload distribution and minimizing thread creation overhead. Our success was not just in achieving a performance boost but also in the insights gained regarding the nuanced trade-offs in parallel computing, affirming the critical need for a harmonious alignment between algorithmic design, data distribution, and hardware capabilities.

4 RESULTS AND DISCUSSION

4.1 KD Tree

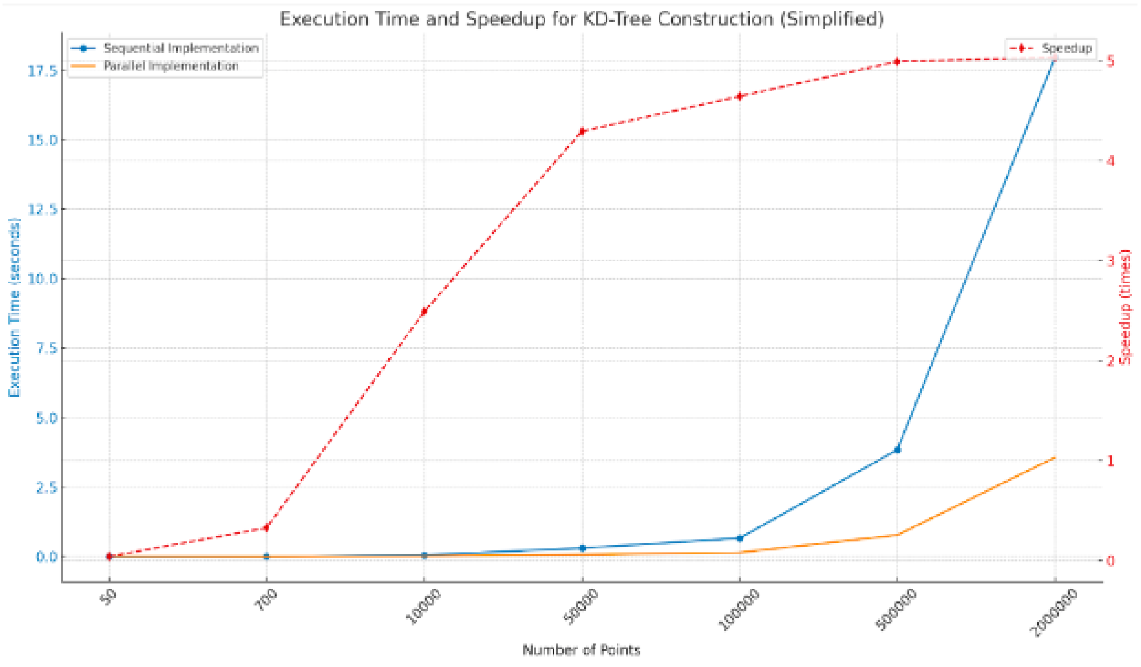


Figure 4: Speedup on GHC machines using 8 cores on various datasets

The graph illustrates the performance gains achieved by a parallel implementation of a computational task across various dataset sizes. Notably, the speedup achieved becomes more pronounced as the dataset size increases. For smaller datasets, the overhead of parallelization does not offset the benefits, resulting in a speedup less than one. However, as the dataset size grows, the efficiency of parallel processing becomes evident. For example, with 10,000 points, the parallel implementation is about 2.49 times faster than the sequential one. This trend continues, reaching over 5 times speedup with a dataset of 2 million points. These results underscore the scalability of parallel computing, where significant performance improvements are realized as the computational load increases.

4.2 KNN

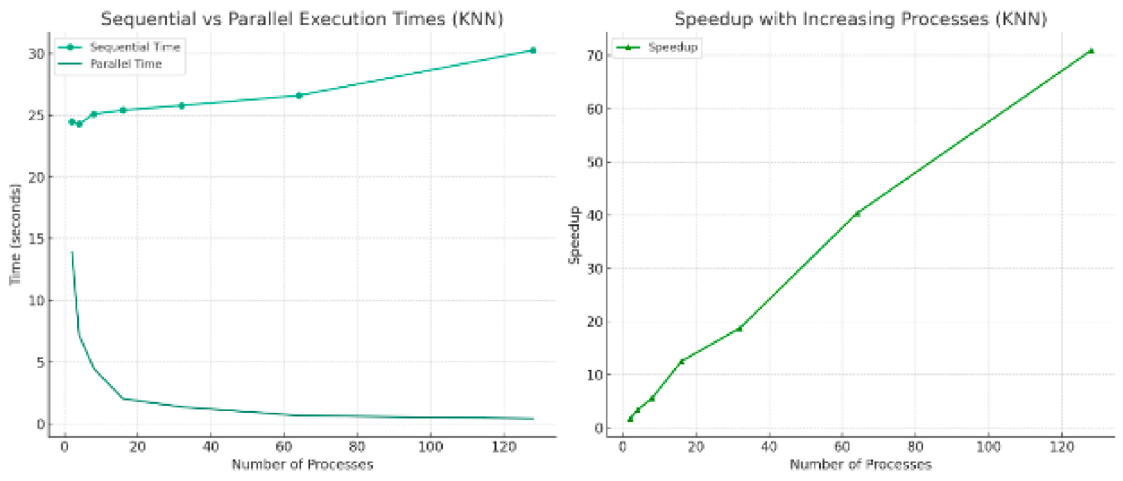


Figure 5: PSC machines; Dataset of 2 million points; searching for 1000 neighbors

The left graph compares execution times of the kNN algorithm running in sequential mode versus parallel mode as the number of processes increases. The sequential time remains constant at around 26-33 seconds. The parallel time decreases sharply from around 26 seconds to below 5 seconds as the number of processes increases from 2 to 32, showing a significant reduction in time with more parallel processes.

From 32 to 128 processes, the parallel execution time slightly increases, suggesting a possible overhead due to increased communication or synchronization requirements in the parallel algorithm, or a saturation point where adding more processes doesn't yield further benefits.

The right graph shows the speedup factor achieved by the parallel kNN algorithm compared to its sequential counterpart. The speedup increases rapidly with more processes, starting from a factor of about 1.7 with 2 processes and reaching over 40 with 128 processes. This indicates that the parallel algorithm is efficiently utilizing the additional processes to reduce the overall execution time.

There is a non-linear relationship between the number of processes and the speedup factor, suggesting that the algorithm scales well with increased parallelism up to a certain point. The slight increase in parallel time beyond 32 processes and the non-linear speedup factor suggest that there are diminishing returns as more processes are added, which is common in parallel computing due to overhead and the finite capacity of computational resources.

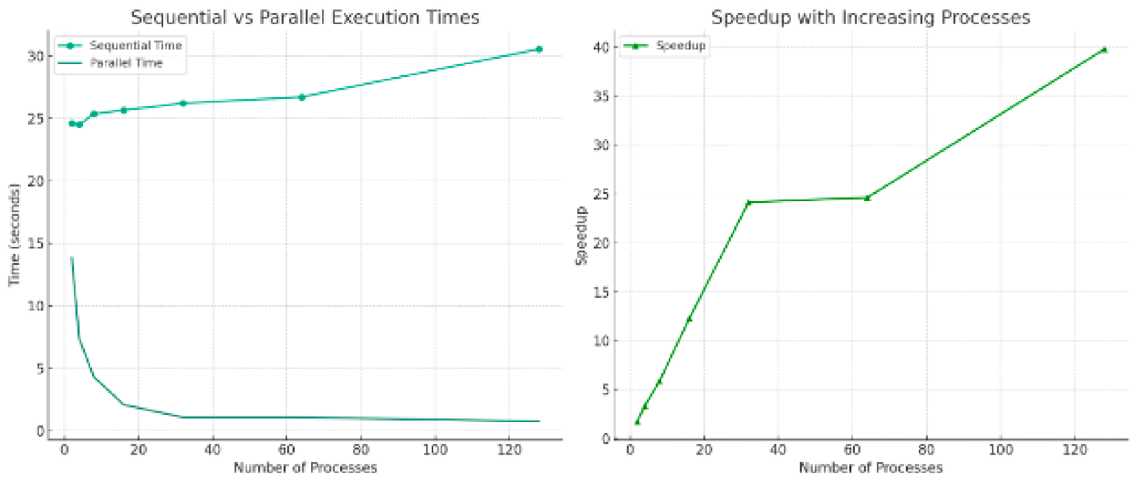


Figure 6: PSC machines; Dataset of 2 million points; searching for 10,000 neighbors

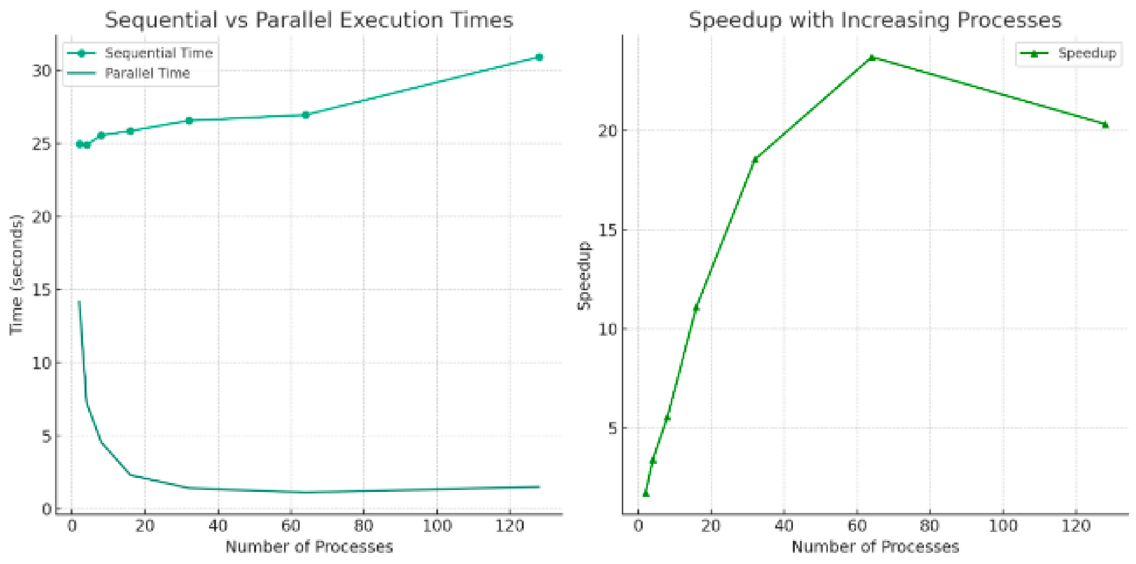


Figure 7: PSC machines; Dataset of 2 million points; searching for 50,000 neighbors

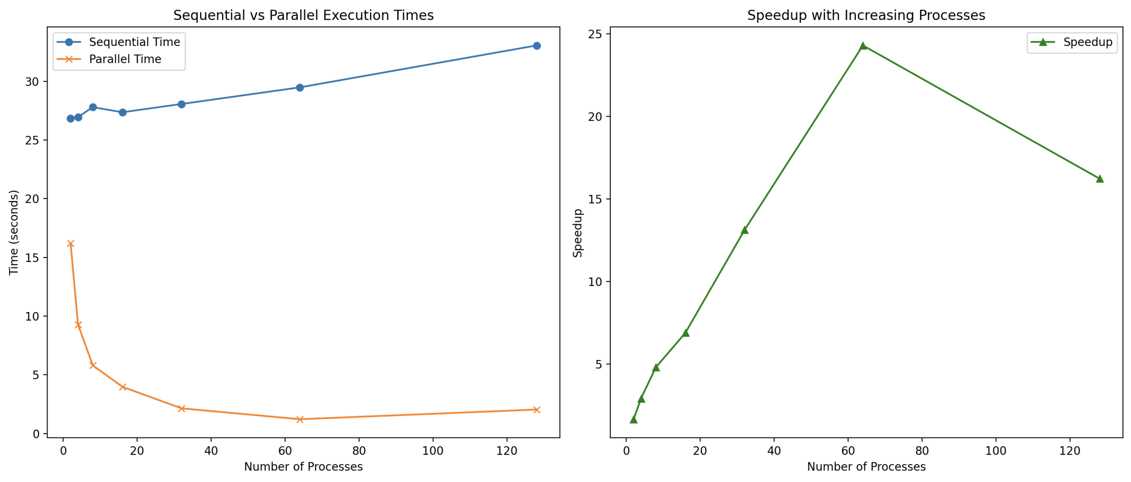


Figure 8: PSC machines; Dataset of 2 million points; searching for 100,000 neighbors

The sequential execution time remains relatively constant across different process counts, as expected, because it does not benefit from additional processes. The parallel execution time decreases significantly as the number of processes increases, showing an improvement in performance due to parallelization. However, the time reduction starts to plateau after a certain number of processes, indicating possible overheads or limitations in further performance gains.

Initially, the speedup increases steeply as more processes are added, which suggests that the algorithm parallelizes well and can effectively use additional resources. Beyond a certain point, the speedup curve starts to flatten and even decreases, highlighting a non-linear relationship and suggesting diminishing returns with the addition of more processes.

The decrease in speedup at higher process counts, such as with 50,000 points and 100,000 points with 128 processes, can be attributed to several factors:

Cache Coherence Issues: As more processes are involved, maintaining cache coherence becomes more challenging. The overhead of keeping the data consistent across multiple caches can degrade performance, especially when the working set size of the data exceeds the cache size.

Communication or Synchronization Overhead: With many processes, the cost of coordinating work between them (such as through barriers and locks) can outweigh the benefits of parallelism. This is particularly true for algorithms like kNN, which may require frequent data sharing and updates.

Data Transfer Bottlenecks: The speed of data transfer between the memory and processors can limit performance. Similarly, if data need to be transferred across a bus, the bus bandwidth can become a bottleneck.

Load Imbalance: If the distribution of work is not even across processes, some may finish their tasks while others are still working, leading to idle time and reduced speedup.

Algorithmic Limitations: Inherent algorithmic dependencies can limit parallelism. For kNN, if the search for neighbors requires traversing the same regions of the KD-tree, this can create contention and limit the effective parallelism.

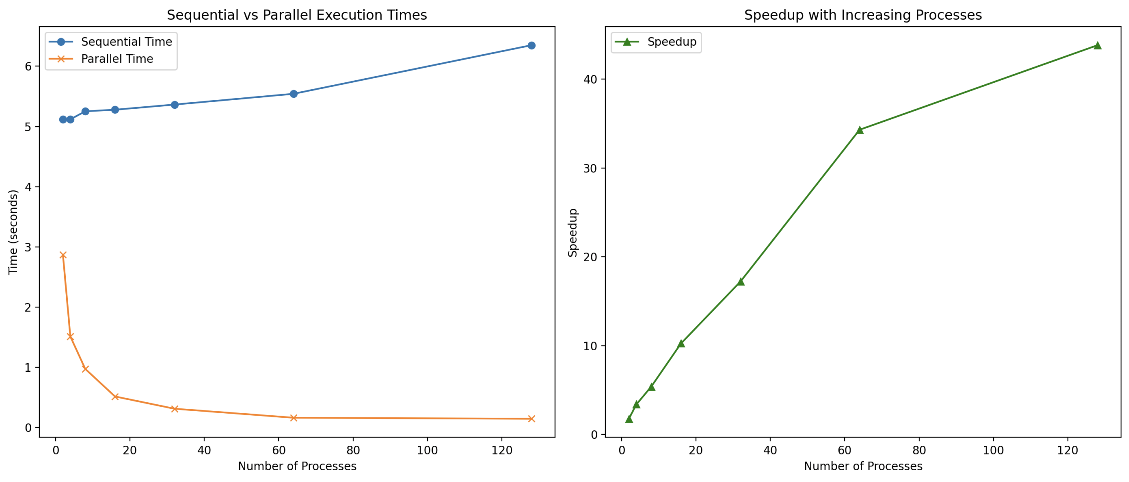


Figure 9: PSC machines; Dataset of 500,000 points; searching for 1000 neighbors

The left graph demonstrates a significant decrease in execution time when the process count increases from 2 to 32, suggesting effective parallelization for smaller numbers of processes. The parallel execution time stabilizes beyond 32 processes, which could be due to several factors such as overheads in managing a larger number of processes, memory bandwidth limitations, or suboptimal workload distribution.

The right graph shows a steep increase in speedup up to about 16 processes, indicating that the algorithm scales well with additional processes up to this point. Afterward, the speedup growth rate diminishes, which might be due to the reasons mentioned above, such as synchronization overhead or the algorithm reaching the limits of its parallelizable components.

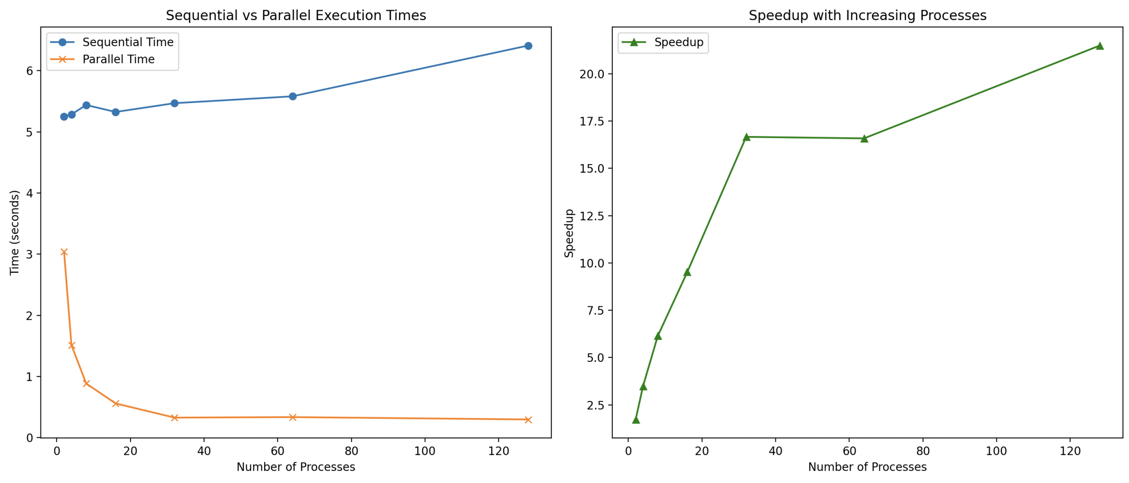


Figure 10: PSC machines; Dataset of 500,000 points; searching for 10,000 neighbors

For 10,000 neighbors, the pattern is similar, with the parallel execution time decreasing as the number of processes increases, but the tail-off in performance improvement is more pronounced. The larger number of neighbors to search could introduce more complexity in the computation, potentially leading to increased communication overhead and reduced effectiveness of parallelism at higher process counts.

The speedup graph for 10,000 neighbors also shows a rapid increase initially but begins to plateau sooner than in the 1,000 neighbors case. This suggests that while the algorithm benefits from parallel processing, there are inherent limitations, possibly due to data dependencies or the increased complexity of maintaining a larger list of nearest neighbors.

Possible Causes/Reasons for the behavior:

Cache Coherence: As more processes work concurrently, the overhead of maintaining a consistent view of memory increases, especially with a larger search space (10,000 neighbors).

Communication Overhead: With more neighbors to search, there's potentially more data to share between processes, which can slow down the overall performance.

Load Balancing: If the workload is not evenly distributed, some processes might be idle while others are overloaded, leading to inefficiencies.

Memory Bandwidth: Searching for more neighbors requires more data to be read from and written to memory, which could become a bottleneck if the memory subsystem can't keep up with the processor speeds.

Algorithmic Scalability: The kNN algorithm may have parts that do not scale linearly with the number of processes, leading to sublinear speedups.

In conclusion, our results show that while parallel processing can provide significant performance benefits, there are practical limits to scalability. These limits can be due to a variety of factors, including but not limited to those mentioned above.

4.3 Final Results and Discussion

In the realm of parallel computing, our comprehensive analysis, encompassing the construction of a KD-tree with OpenMP and the parallelization of kNN using both OpenMP and MPI, provides a nuanced understanding of the scalability and performance bottlenecks associated with these approaches. Our experimental results demonstrate a clear advantage of parallel processing in reducing execution times, particularly notable when the dataset size is large. For instance, with a dataset of 2 million points, the parallel kNN search yielded a speedup of over seventy times (with 128 processes) compared to the sequential approach.

However, this performance gain is not without limitations. We observed that the improvement in execution time begins to plateau and even slightly decreases as the number of processes becomes fairly high. This phenomenon suggests the emergence of overheads possibly linked to communication, synchronization, and cache coherence, especially prevalent when the algorithm scales to a vast number of processes. Additionally, the speedup factor, while increasing rapidly with more processes initially, exhibits a non-linear relationship, plateauing and decreasing beyond certain thresholds. Such behavior indicates diminishing returns due to factors like load imbalance, data transfer bottlenecks, and algorithmic dependencies, which hinder further gains from parallelization.

Our findings with smaller datasets, such as 500,000 points, reinforce these observations. The parallel execution time significantly decreases with an increasing number of processes up to 32, after which it stabilizes, highlighting the efficacy of parallelization for a moderate number of processes. Yet, as we extend our search to a larger number of neighbors, the complexity of the computation increases, potentially leading to enhanced communication overhead and reduced parallelism effectiveness at higher process counts.

The behaviors observed across different datasets and neighbors searched point to a common set of underlying challenges in parallel computing: the need for efficient cache coherence mechanisms, the balance between computation and communication overhead, the criticality of load balancing, and the nuances of algorithmic scalability. These insights emphasize the importance of careful algorithm design and optimization in parallel environments to mitigate the diminishing returns observed with increased parallelism, ensuring that the performance scales with the addition of computational resources.

REFERENCES

[1] V. S. Rekha (2020). All About KNN Algorithm. *Medium*. Available at https://rekhavsrh.medium.com/all-about-knn-algorithm-6b35a18c2b15.

[2] J. L. Bentley (1975). Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9), 509–517.
Available at https://dl.acm.org/doi/pdf/10.1145/361002.361007.

[3] J. H. Friedman, J. L. Bentley, and R. A. Finkel (1977). An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, 3(3), 209–226.
Available at https://core.ac.uk/download/pdf/19529722.pdf.

[4] C. Kingsford and S. Salzberg. KD-Trees for Semi-dynamic Point Sets. Lecture Notes, Carnegie Mellon University.
Available at https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf.

[5] D. Zhang, M. M. Islam, and G. Lu (2008). A KNN Model-Based Approach in Classification. In OTM Confederated International Conferences "On the Move to Meaningful Internet Systems".
Available at https://www.researchgate.net/publication/2948052_KNN_Model-Based_Approach_in_Classification.

5 WORK DISTRIBUTION

Doreen Valmyr: 54%
Neelansh Kaabra: 46%