Assignment 2 – Report

Submitted by: Fatma Sayed Mahmoud Sayed Moustafa

ORFEO username: fmoustafa

Introduction

KD-trees are data structures that partition space according to certain conditions, hence it organizes points in a k-dimensional space. Their application ranges from nearest neighbour searches (KNN) for information retrieval to astronomy. A KD-tree is a tree which mainly compares a single dimension cycling through the other dimensions, this process is done one dimension at a time thus keeping the balance of the tree. At each iteration i, dimension d is selected to do the bisection of data along it. Thus having a right subtree and left subtree. In the next iteration, one of the other dimensions is selected to perform the bisection along it, creating right and left subtrees. This procedure is done along all the dimensions and then repeated iteratively. A pivot point p is found such that all points in a given axis greater than p is to the right of tree forming the right subtree, and all oint less than the pivot is to the left of the tree forming the left subtree. In this study, we focus on 2-D trees, thus the bisecting is done along the x-axis in one iteration then along the y-axis in the next iteration and so on. Also, the pivot point found in each iteration is going to be the median; since data are homogenously distributed in every dimension.

Algorithm

In this implementation of the KD-tree, two conditions were considered:

A1- Immutable data set: neglecting insertion and deletion operations

A2- Data points homogenously distributed in all k-dimentions

Building the Kd-tree consisits of these steps:

1- Defining of the splitting dimension/axis which will change for each iteration
2- Defining the node structure of the kd-tree which has the following attributes:
    a. splitting dimension: round robin through dimensions
    b. splitting element
    c. right and left sub-trees

    nodes with no children (left and righ subtrees are null) contains only a data point = leaf

3- calculating pivotal point
4- constructing left subtree
5- construction right subtree

OPENMP steps:

1- Define structure to represent the nodes of the KD-tree
2- A check is made to ensure that there is at least 2 points present in the chosen direction
3- Data is sorted along the spliting dimension
4- The median is calculated (pivot or root of the subtree constructed)
5- Datapoints are shared among parallel workers(threads)
6- Left and right subtrees are constructed given the median

To include an MPI parallelization strategy:

1- MPI threads, rank and number of processors are initialized
2- Data is sorted along splitting dimension d1 for process of rank=0 (Master)
3- Median is calculated and right and left subtrees are constructed
4- Right subtree is sent by process 0 to process 1 by use of MPI_Send on process 1 side and MPI_Recv on process 1 side
    a) Data is sorted along splitting dimension d2
    b) Median is calculated and right and left sub-sub-trees are constructed
    c) Right sub-sub-tree is sent to process 2
    d) Left sub-sub-tree remains for process 1
5- Left subtree remains for process 0
    a) Data is sorted along splitting dimension d2
    b) Median is calculated and right and left sub-sub-trees are constructed
    c) Right sub-sub-tree is sent to process 3
    d) Left sub-sub-tree remains for process 0
6- This procedure continues till we divide all the data among the number of processors we have
7- In each process, the data is handled with the openMP parallel threads to construct the kd tree
8- MPI_Finalize to end the MPI parallelization phase

Implementation

In this study, the KD-tree was implemented on c++11 and on orfeo thin nodes as well as GPU nodes. Different sizes of datasets were tested starting with 1000 points till 100,000,000 points incrementing by powers of 10. Only 2 dimensions were considered. 2 codes were tested: serial KD-tree and parallel OpenMP tree. A trial on implementing a hybrid OpenMP/MPI code was made, however, it was not successful.

To sort the datapoints, nth_element was use. It is a partial sorting algorithm that arranges elements from the first to last. It does not sort the data, however, the element in the nth position is the element that should have this position if the values were sorted. Since in our tree, we are more interested in the splitting of values less than the pivot or greater than the pivot, this partial sorting approach is effecient. It is asymptotically faster than other sorting approaches and performs well for large problems.

In the implementation of openMP and during the building of the tree, #pragma omp commands were used.

1) #pragma omp parallel shared (...) was used to have shared variables
2) #pragma omp task to construct left and right subtree

Performance model and scaling

To measure the performance of the parallelization of the KD-trees: strong and weak scalability were measured. Time of computation was measured in each run. Speed-up and effeciency were calculated using the following formulas:

- Speedup is the ratio between time to execute the serial workload and time to execute the workload parallelly with P parallel workers

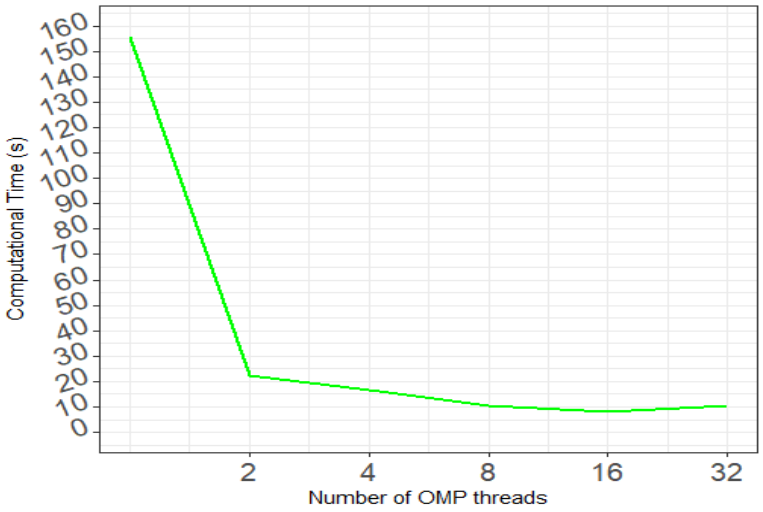
$$S(P) = \frac{T(1)}{T(P)}$$

- Efficiency is the average speedup by one parallel in a parallel program

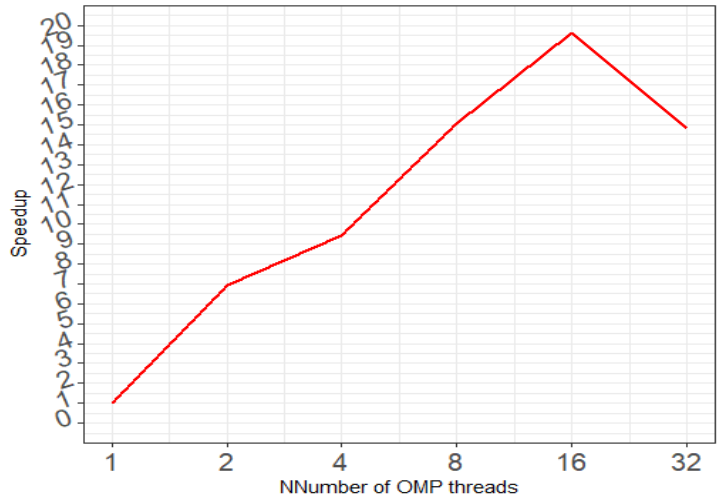
$$E(P) = \frac{S(P)}{P}$$

- Strong scalability
  Strong scalability measures the performance of the parallelization when the number of parallel workers increase while keeping the problem size constant.
  - Test cases and iterations:
    - 2, 4, 8, 16, 32 OMP threads for 100,000,000 datapoints
  - Results and conclusion for 100,000,000 datapoints:

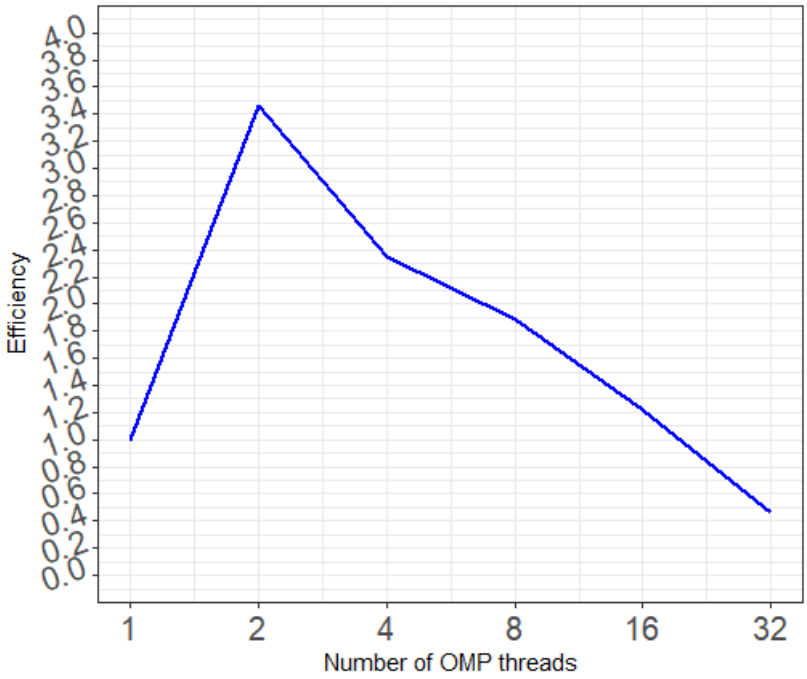
  The below figures shows that the computational time decrease tremendously from the serial case to the parallel case. With number of threads ranging from 2 to 16 increasing quadratically the time decreases. It could be concluded that increasing the number of threads further will not decrease the time that much.



The speedup gains from using N threads are illustrated which increases with the number of OMP threads. Of course, for the case of having 32 OMP threads since the time increases, the speed up decreases.

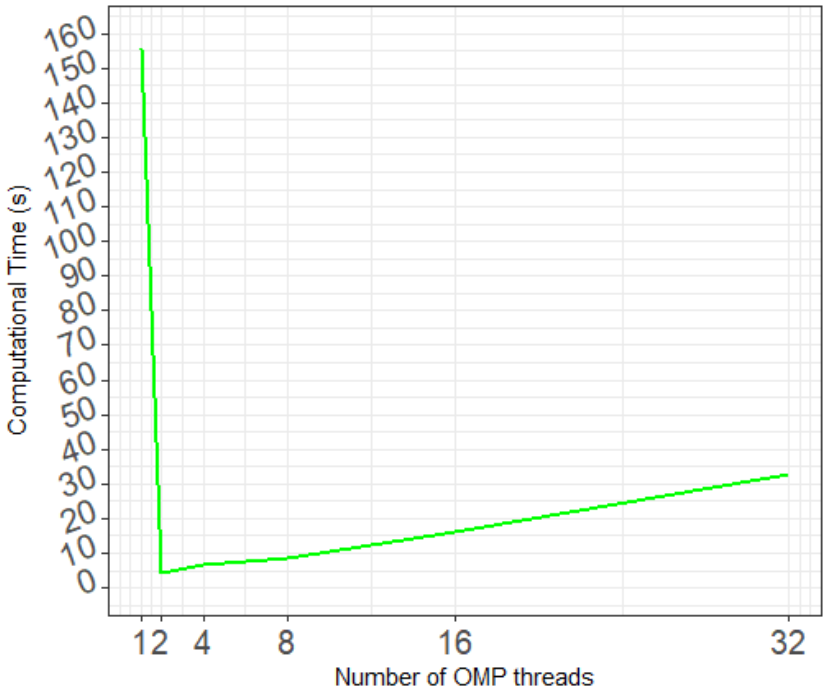


The efficiency increases from the serial case to have 2 OMP threads. Then as we increase the OMP threads , it decreases to values lower than the serial case.

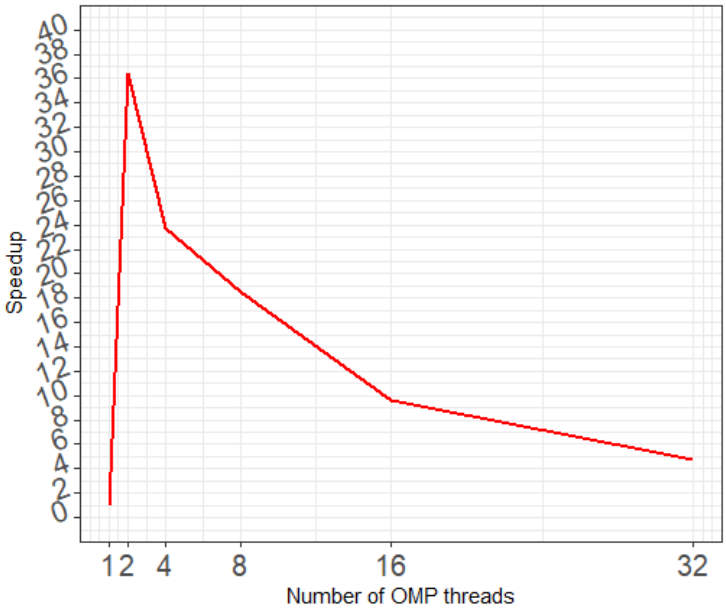


- Weak scalability
  Weak scalability measures the performance of the parallelization when the number of parallel workers increase with increasing the size of the problem by a factor equal to the number of parallel workers.
  - Test cases and iterations:
    - 2, 4, 8, 16, 32 OMP threads for N*P dataset, where N is the 10,000,000 datapoints and P is the number of
  - Results and conclusion:

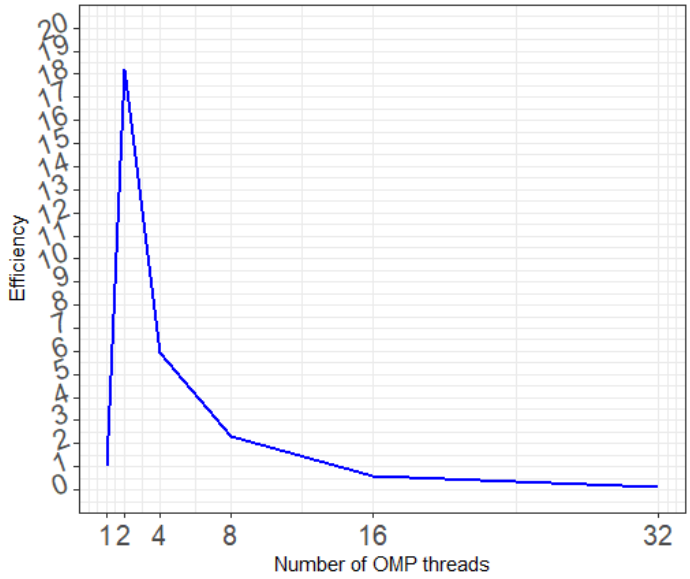
The below figures shows that the computational time decrease initially when going from the serial case to parallel. However, it shows an increasing behavior as we continue to increase the threads and problem size.



By observing the speedup, it is noticed that it decreases as we increase the number of threads.



However, the efficiency decreases with the increase in the number of threads and increase in problem size.



Discussion

- As mentioned before, an unsuccessful attempt of implementing an MPI code was made. With further work and trials, a solution could be reached.
- By using a different sorting approach, the performance might be enhanced