Final report

Bruno Bonaiuto Bolivar

Introduction
the assignment consists in a implementation of a kd-tree which is a space-partitioning data structure for organizing points in a k-dimensional space in this specific implementation the data will be taken from a vector of data points

since the objective of the implementation is to perform this data structure in parallel programming by the OpenMP, in that way a collection of data its able to measure the performance and provide some conclusion

Algorithm
The kd-tree is a data structure presented in the 70's to represent a k-dimensional space performing an efficient way of searching. the basic concept is to compare against 1 dimension at a time per cycle through the dimensions to have a balanced kd-tree, at every iteration it creates a left sub-tree and a right sub-tree

Assuming the kd-tree made from:

- inmutable data set
- homogeneously distributed data points among the k dimensions

there are some assumptions to build a kd-tree:

- the median of the vector is given by the according dimension
- the vector is split with the median in the given dimension
- the root node is assigned as the median index
- the algorithm is applied recursively, with the left points and the right points becoming the new data points

Implementation
the kd-tree implementation was made with c++11, the approach followed was an OpenMP

For the OpenMP version of the program, a recursive approach has been chosen.

kd-Tree_openMP: snippet

```
kdnode{
   Int points;
   Struct kdnode *left, *right;

knode* build_kdtree(vector, axis, leaves){
   if(vector_size = = 1){
      //create a single node
   }else{
         int median;
         int vector_size;
         if(axis = = x_axis) {
            //sort the vector
         }else{  // axis = = y_axis
               //swap the vect
```

```
                //sort the vector swapped
                //swap the vector one more time
            }
            //kdnode_left  //computed by a thread
            //kdnode_right //computed by another thread
        //return node
};
```

From the kd-tree openMP it's observed a split point according to left and right, which is perform in a parallel region. The algorithm is done by using OMP tasks.

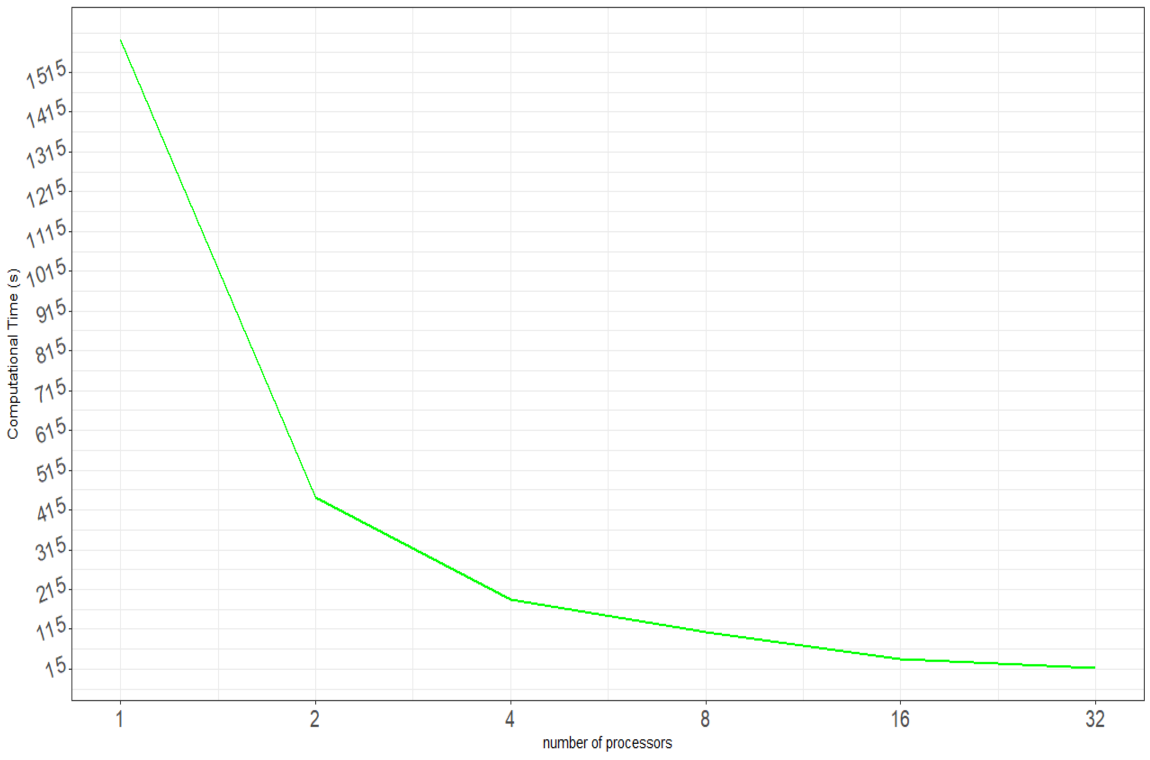
The OpenMP implementation of the algorithm is simple, the syntax is done by adding the directives for the serial version of the kd-tree to perform it in parallel.

The parallel region is created by the master thread, the first node is created and then for the sorting and swapping are done in a shared parallel region, then the nodes left, and right are created using openMP task.

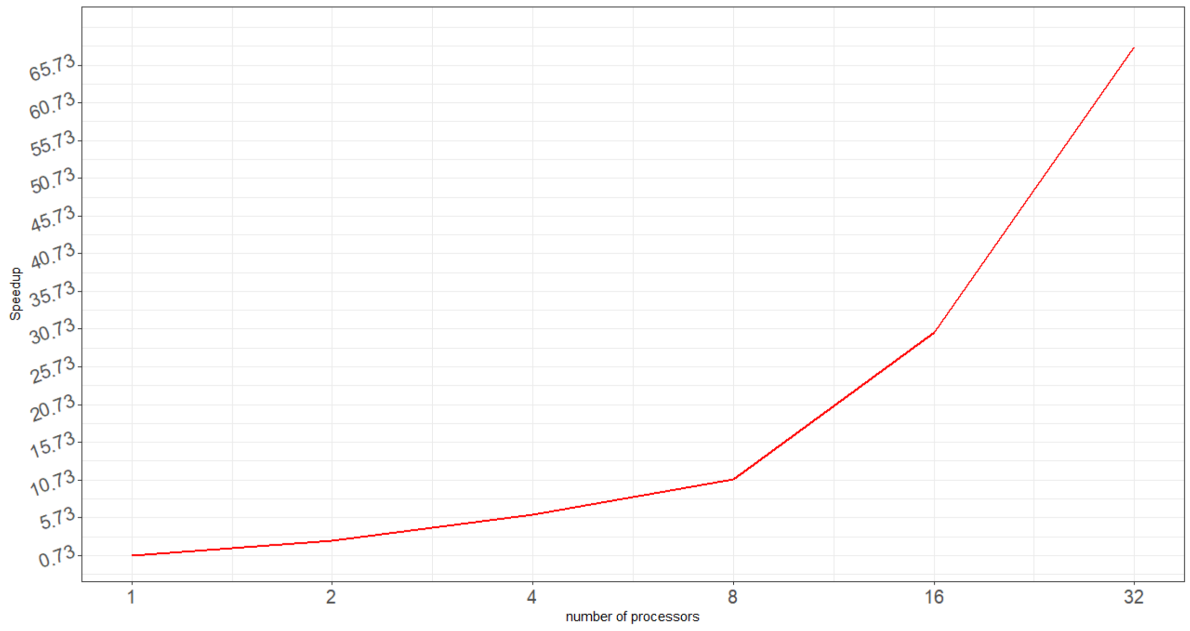
The thread that created the task is free, and it will perform the next task recursively. At the end the parallel region is closed, and the kd-tree is created.

Performance
In this section it be discussed the performance from the execution time of the parallel kd-tree,

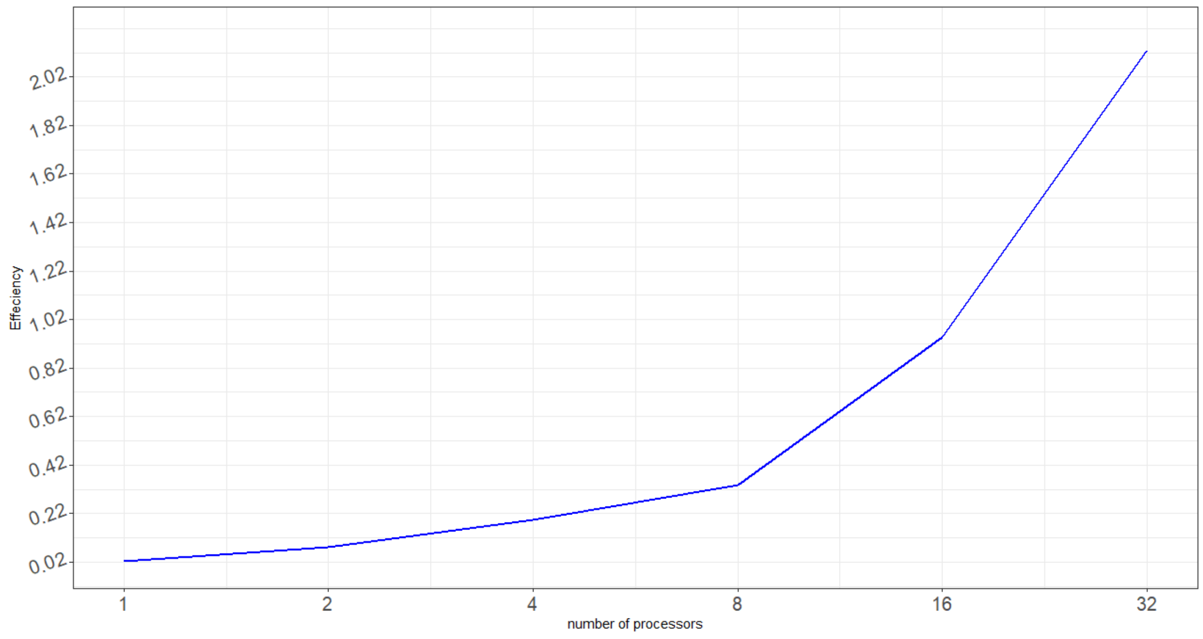
<u>Strong scalability</u>
The **strong scalability** is done by performance with an increasing number of processors, with the same data. The computational time decreases



The **Speed-up** for **strong** scalability while the number of processes increase the time increase

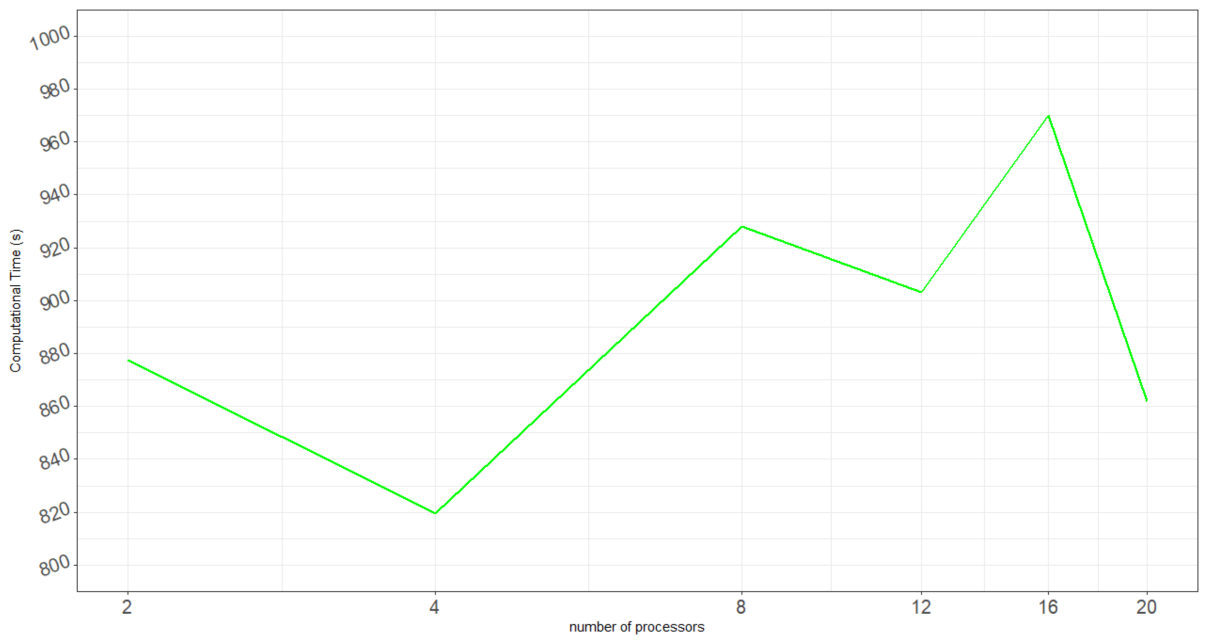


The **efficiency** for the **strong** scalability as we increase the number of processor time increases

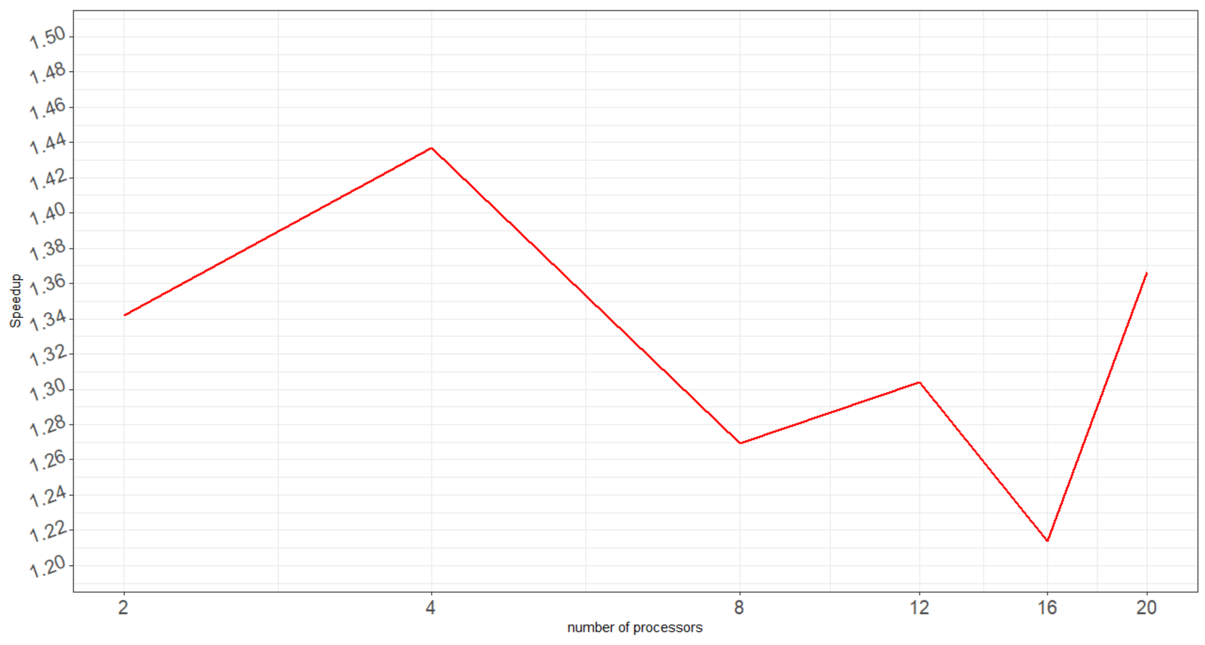


Weak scalability

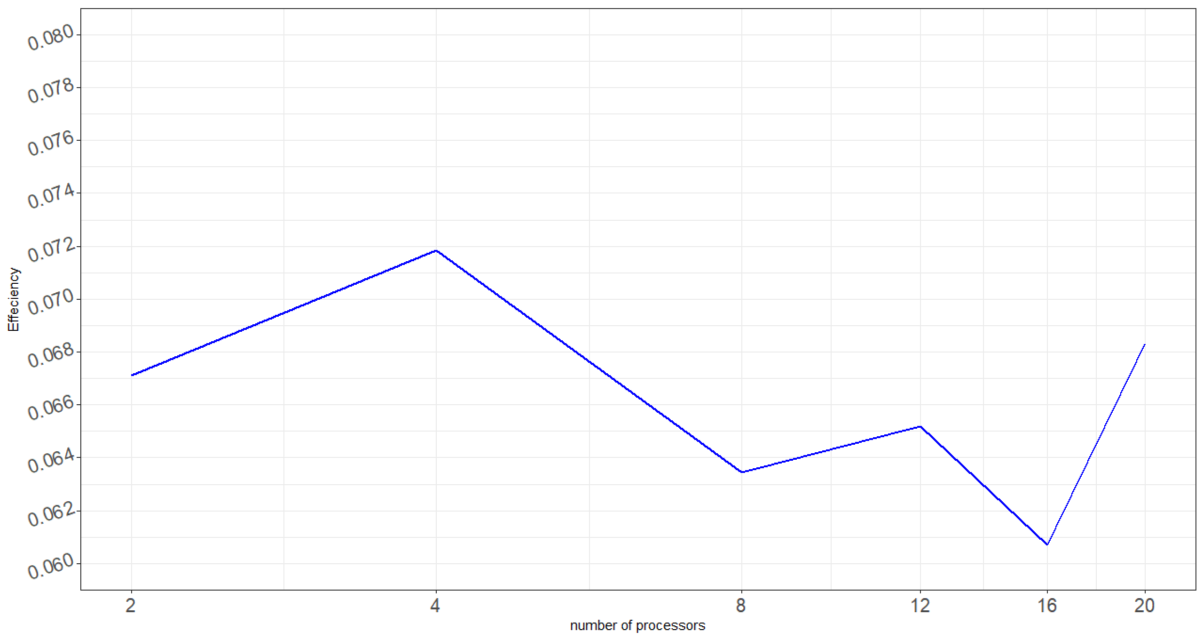
the **weak scaling** is done by a scaled problem size with respect to the number of processors, the computational time behave as a constant time.



The **Speed-up** for **weak** scalability the times remains as a constant also



The **efficiency** for the **weak** scalability behaves also in the same way as the speed-up



Discussion

The weak scalability is done by a fixing problem size per thread, and after increasing the size sequently, it behaves as a constant. For both cases in strong and weak scalability shows and no-linear increasing.

Possible improvements on the code

The code could be improved by writing a better code optimizing on the implementation, since the recursive part sorts the data which takes a cost, by avoiding this part the code should be run quicker.