# Report Assignment 2

Verardo Thomas

18/2/2022

## Introduction

The aim of this report is to explain how a KD-Tree is implemented in a serial or parallel way.

In the serial part, the algorithm is constructed as a "normal" program, without keeping in consideration how the processor will works. In the parallel part, it will be consider how the processor can distribuite the amout of work among processes or threads. To implement this part, it was used two famous parallel programming library, that are **OpenMP** and **Open MPI**.

It's important to make a premise. In the following report, where there is written "process", it means either MPI process or OpenMP threads. This was done to avoid confusion in the performance paragraph.

## Algorithm

A KD-Tree is a data structure presented originally by Friedman, Bentley and Finkel in 1977 to represent a set of k-dimensional data in order to make them efficiently searchable. A KD-Tree is a tree whose basic concept is to compare against 1 dimension at a time per level cycling cleverly through the dimensions so that to keep the tree balanced. At each iteration $i$, it bisects the data along the chosen dimension $d$, creating a "left-" and "right-" sub-trees for which the 2 conditions $\forall x \in sub-tree, x_i < p_i$ and $\forall x \in sub-tree, x_i > p_i$ hold respectively, where $p \in D$ is a point in the data set and the sub-script $i$ indicates the component of the $i-th$ dimension.

To build the tree, two assumptions were made:

- The dataset, and hence the related KD-Tree, it's immutable;
- The data points are homogeneously distribuited in all the $k$ dimension, i.e. the data are taken pseudo-randomly from the function *rand()* of the *random* library of C++.

Since that can be assume that the data are homogeneously distributed in every dimension, it was decided to choose $p$ as the median element along each dimension. In this way, the binary three will be balanced[1].

These are the abstract steps to bulid the KD-Tree:

- First of all, along one dimension, the data points are partially sorted, in this way to find the median.
- The pivot, that is the median, will be the root on the sub-three.
- All the elements that are in the left of the pivot will be the left child of the root and all the elements that are in the right of the pivot will be the right child of the root.
- The algorithm is applied recursively, with the left child points and the right child points that are the new data points of the first step.

[1] A tree is balanced if and only if all of the left and the right subtrees' heights differ by at most one

# Implementation

The KD-Tree implementation was made with C++11 and is written to work with the number of k-dimension $> 0$ (*N_DIM > 0*). The code is a generic programming algorithm, because it's written with C++ templates. The implemented algorithm doesn't use a sorting algorithm, but it uses the function provided by the standard library "*std::nth_element()*". This function is a partial sorting algorithm that rearranges elements in *[first, last)* such that:

- The element pointed at by *nth* (the median) is changed to whatever element would occur in that position if *[first, last)* were sorted
- All of the elements before this new *nth* element (the median) are less than or equal to the elements after the new *nth* element.

With this function, it's possible to omit the implementation of a sorting algorithm because it has the best "worst expected running time", that is O(N).

For the testing and building phase, since the number are supposed to be described by an uniform distribution, it was decided to generate data randomly for all the axis. Each point has the random data and the number of dimensions of the data. Each node of the tree has the point that represent, the axis to which it's applied, the pointer to the left child and the pointer to the right child.

To implement the parallel part, it was decided to use two different methods. Unlike the OpenMP part, in which the code is pretty similar but with the addition of the *#pragma omp* parallel function that are called in the recursion phase, for the implementation of Open MPI, the code (no the algorithm) changed a little bit. In fact, the strategy used to implement this part was to divide for each process, a sub-tree, that that will be merged in only one process.

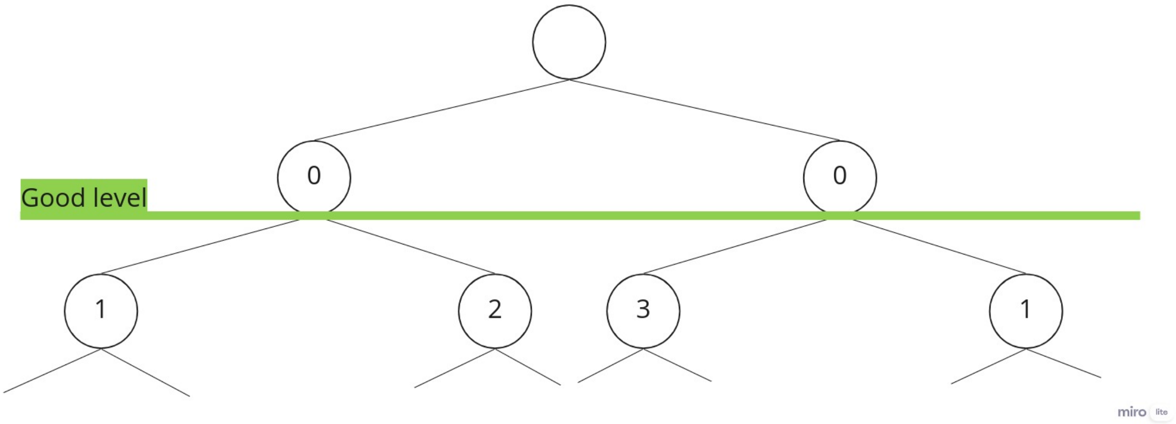


Figure 1: Example of the building process with 4 process

It's important to dived the work of the group (from the process 1 to the process N), that their primary function is to construct a sub-tree and send to the main process, and the work of the single process (the process 0), that his primary function is to merge all the sub-tree and to construct the final KD-Tree. These are the steps of the workers:

- Until a good level of the final tree is reached, the group of processes builds the tree in a serial way. A good level is reached when the half of the number of process involved is equal to the power of 2 to level (so $np/2 == 2^{level}$).
- When you get to to this level, each process works alone creating in a serial way the sub-tree and send it to the master process, with the rank 0. There is a shared variable that assign the work of each process, that is it has to work in the right or in the left. If the number of process is smaller that the number of the sub-tree that has to built, this variable automatically assign which process has to work two times.

Meanwhile, there is the process with rank 0, that does the first part like others processes and then waits to receive the messages with all of sub-trees.

The main problem of this implemented code with Open MPI is that each sub-tree has to be serialized in a string before sending and deserialize the message received after.

# Performance scaling and discussion

## Performance model

First of all, it was created the performance model of OpenMP and Open MPI. The model of the OpenMP part was:

$$T_{OMP} = \frac{T_{serial}(N)}{NP} + k$$

Where $N$ is the total number of points, $T_{serial}$ is the total time to create a KD-Tree of $N$ points with one process, $NP$ is the number of used processes and $k$ is a constant variable referred to the time to load in run-time some OpenMP start-up stuffs.

Instead, the model of Open MPI was different, because it has to calculate also the time to serialize/deserialize. In fact, the model is this one:

$$T_{MPI} = \frac{T_{serial}(N)}{NP-1} + T_{serialize}(\frac{N-NP-1}{NP}) + T_{deserialize}(\frac{N-NP-1}{NP})$$

The time to serialize and deserialize, depends of how much points there are. In fact, the number of points it isn't always $N$, but they are less. This because, the first part of the code is runned by all of the processes and they build the same tree in a "serial" way, until they rich a good level to start to work alone. So, there are $N - NP - 1$ because one part of the KD-Tree is already been built. Another thing is that are used only $NP - 1$ processes for the parallelization, this because the process 0 only have to recieve the KD-Tree, deserialize them and combine them.

If we want to understand much better the time to serialize/deserialize, it has been made another model:

$$T_{MPI} = \frac{T_{serial}(N)}{NP-1} + log(\frac{N-NP-1}{NP}) + NP * k * \frac{N-1}{2} * log(n)$$

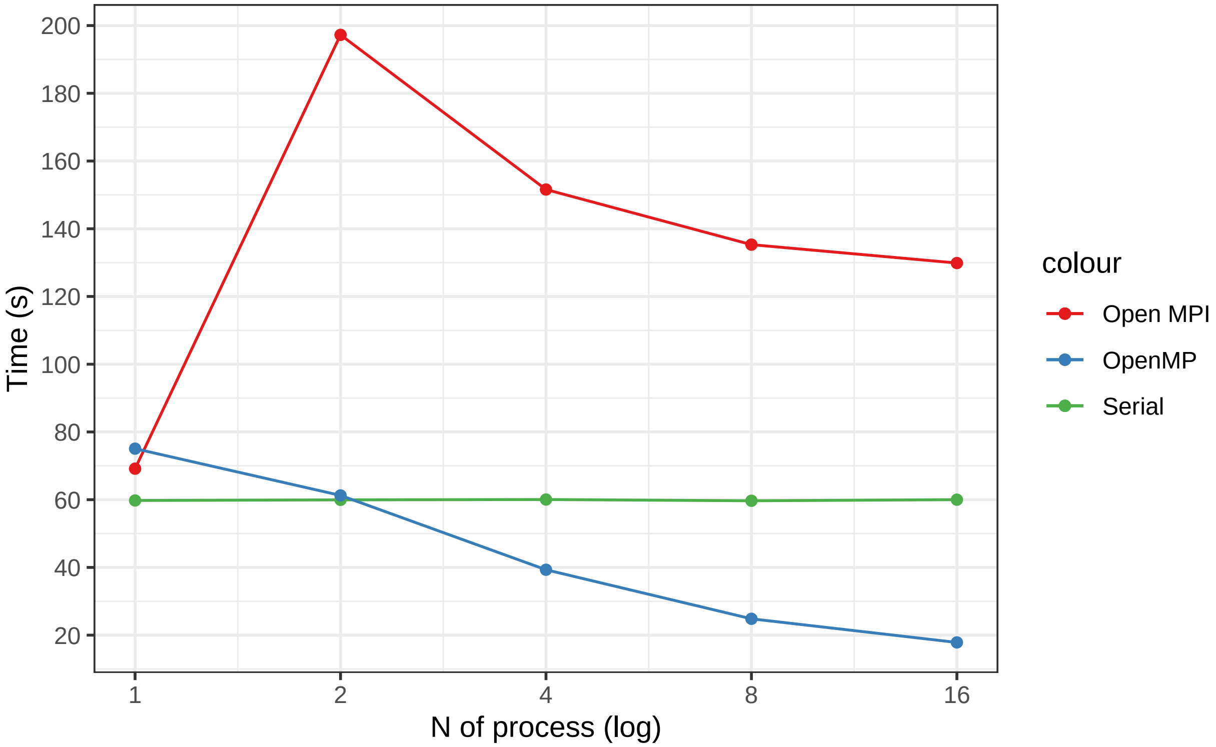
## Real performance

In this section it will presented the real performance of the function responsible for the construction of the tree in serial and in parallel. Theoretically specking, the performance of the parallel code must be better than the serial, because the work is spitted among the processes/thread. All tests have been done in ORFEO GPU nodes. To evaluated the performance of the implemented code, the test was done with $10^8$ data points, each composed to 2 dimension. All the compilation are made with an optimization flags (-O3) because, without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time. Furthermore, the OpenMP part is compiled with some environment variables that are used to maximise the performance. Those are:

- *OMP_PROC_BIND=true* that prevents threads migrating between cores
- *OMP_WAIT_POLICY=active* that encourages idle threads to spin rather than sleep
- *OMP_DYNAMIC=false* that doesn't let the runtime deliver fewer threads than those expected
- *OMP_PLACES=cores*, that set the places to corresponds to the cores.

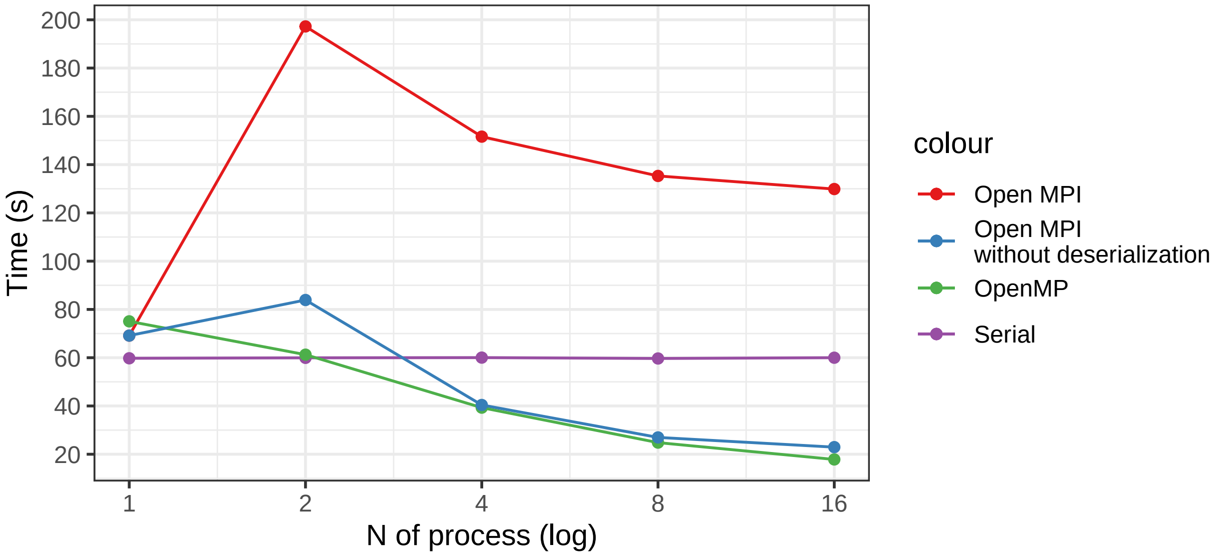
## Strong scalability

The strong scalability explain the performance of the parallel algorithm with the number of processes that increase while the problem size remains constant. The individual workload must be kept high enough to keep all processors fully occupied. The speedup achieved by increasing the number of processes usually decreases more or less continuously.

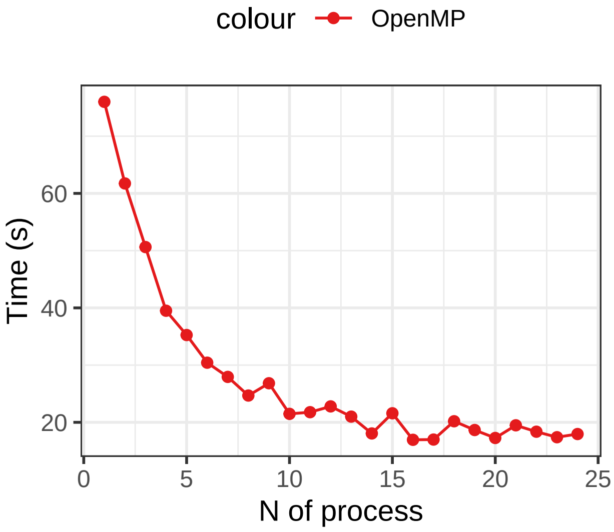


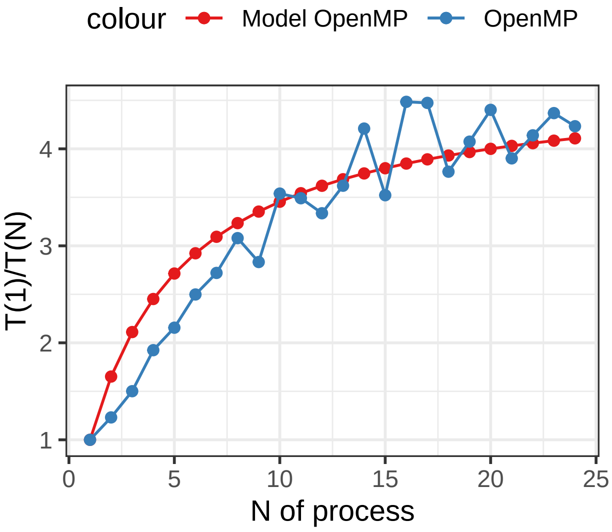
It's possible to see in this graph above that, while OpenMP can paralyzes well, with a execution time close to the half of the serial time, Open MPI can't do a very good job. This is caused by the serialized/deserialized part. Indeed, each processes, after they built their sub-tree, they have to send to the master process, so as to merge all sub-tree together. But, before the message is sent, the sub-tree has to be serialized in a *std::String*. To serialize the sub-tree, the algorithm have to print on a string all levels of that tree. Than, the process with rank 0 has to first receive the message with the serialized tree, deserialize the string and at last, combine with others trees.

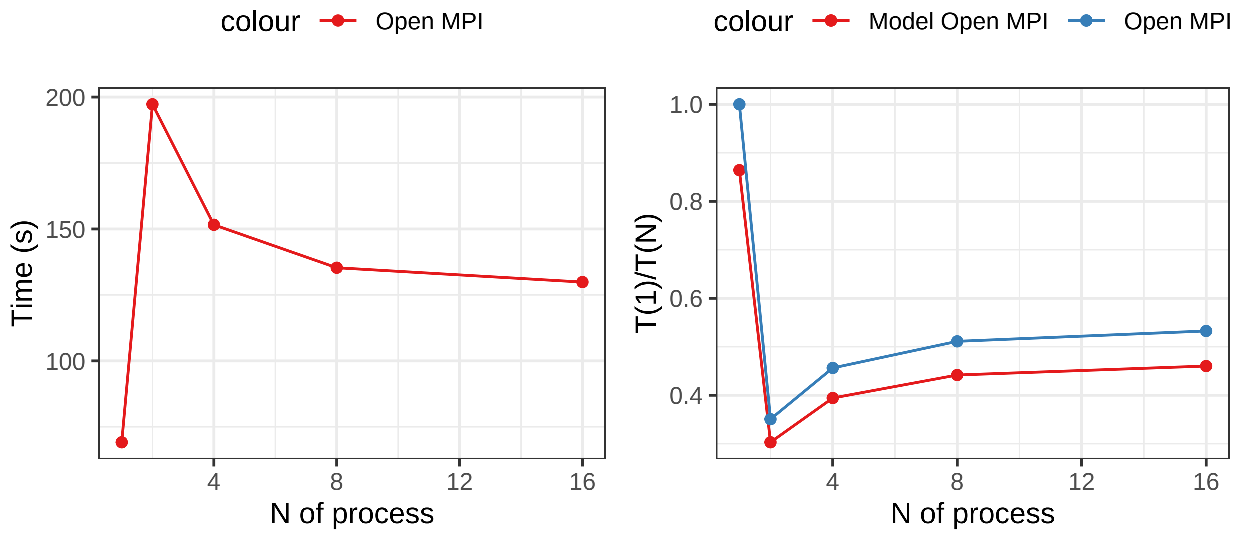
Moreover, the serial execution time of an OpenMP program on one thread it's slower than the equivalent serial program because the compiler may not be able to as aggressively optimize the parallel code as the serial code. In this way, at run-time the code may pay a penalty in start-up performance of instantiating the OpenMP run-time stuff even though the code only uses one thread. The run-time doesn't know this in advance.



After doing some tests, the time to serialized a tree is $s/np$, where $np$ is the number of processes used and $s$ is a constant, that is the average time to serialize the tree **with one process**, and is equal to $72s$ ca. For example, when we use 4 processes, we have the average time to serialize equal to $26s$ ca for each process. But the real problem is to transform the string in a tree; to make this possible, the algorithm have to iterate some loops and this will decrease the total performance. Exactly, the average time in seconds to deserialize is $d/np$, where $np$ is the number of processes and $d$ is a constant equal to $108s$ ca. Hence the deserialization part has been processed by the same process (process 0), if there are more total process, the time of the deserializetion will be less because the it will deserialize less points. To see how much the deserialization affects to the total performance of the Open MPI implementation, it has been done this chart above, where it's possible to see that the performance improves by the constant $d$, that is $265s$ ca. Furthermore, the algorithm of the Open MPI part does a great job when it has to paralyzes the algorithm, because the execution time dicrease from $400s$ with one process to less than $100s$ with eight processes.







To looks better the performance of the OpenMP, the following graph was made (above). In the plot to the left it's possible to see how much time does it takes to build the KD-Tree with two dimension. Notice that, after some processes used in parallel, the algorithm cannot decrease the time to build the tree. In the left, there is the plot that describe the "parallel speedup". The speedup gained from applying N CPUs, Speedup(N), is the ratio of the one-CPU execution time to the N-CPU parallel execution time: $Speedup(N) = T(1)/T(N)$. Intuitively, this ratio should be equal to the number of CPUs, but this never be achieved. This happens because some parts of a program can't be executed in parallel. This phenomenon is describes by the Amdahl's Law, that is:

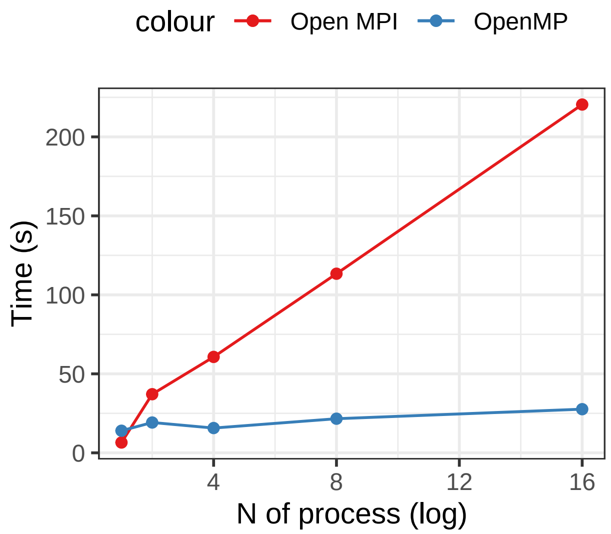
$$S(N) = \frac{1}{(1-P)+(P/N)}$$

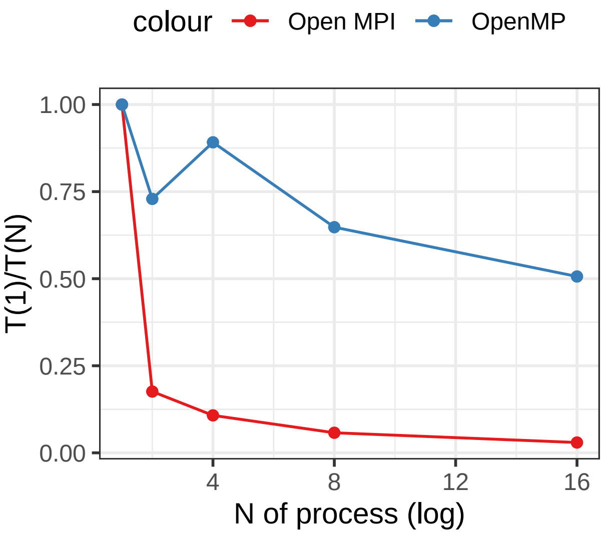
Where $S(N)$ is the maximum speedup that can be achieved using N processes, $P$ is the proportion of the program that can be made parallel and $1-P$ is the proportion that remains serial.

Summing up, we can see that OpenMP has best performance of Open MPI, but if the serialization/deserialization part is done differently, the Open MPI algorithm has greater room for growth.

## Weak scalability

In case of weak scalability, both the number of processes and the problem size are increased. Weak scaling is mostly used for large memory-bound applications where the required memory cannot be satisfied by a single node.





Here there are two plots to measure the weak scalability. They are created with $10^7$ original data points and multiplied by the number of used processes. As already mentioned above, OpenMP is much better than Open MPI, because the time to build the KD-Tree with 2 dimension is always the same. Indeed, in the plot to the left, there is the “weak scaling efficiency”, which calculate the efficiencies as the number of processes increases. The ideal efficency is always 1, where, as the number of processes increase, the time remains constant. But the real efficiency shows a decrease in efficiency with increasing number of processes.

## Future implementation

As it was mentioned before, the main problem of the parallel algorithm in the Open MPI part was the serialization, but specially the deserialization. In fact, the deserialization function is a recursive function but with some loops that degrades performance. To improve this part, it’s possible to improve this function, removing the loops or parallelize them. Another possible solution is removing the entire serialization/deserialization part and, instead of sending the entire sub-tree to the master process, send only a part with his size or send another type instead to the string (std::string is very heavy).

Another mistake that was made, was to add, at every iteration of the function “build_kdtree”, two std::vector that corresponds to the points for the left child and the points for the right child of each node. To improve this function, it was better to use another data type or, like in the QuickSort algorith, to use two integer variables that marked the beginning and the end of where the node would go to see to build the left and the right child.

Furthermore, it was also possible to implement an hybrid code, that takes the advantages of both of the libraries to combine together in one algorithm. Since the main problem is the serialization and the deserialization of the three, it’s possibile to make parallel this part with OpenMP.

## Bibliografia

https://www.archer.ac.uk/training/course-material/2017/08/openmp-ox/Slides/L09-TipsTricksGotchas.pdf

https://hpc-wiki.info/hpc/Scaling_tutorial