Assignment 2

Foundations of HPC 2021-2022

Francesco Ortu | francescortu@live.it |
github.com/Francesc0rtu/HPC-2021-2022-Assignment-2

Introduction 1

Algorithm 2

Implementation 3

Send a tree using MPI 4

Divide work and re-building the tree 4

Full implementation 6

Performance model and scaling 6

Performance model 6

Results 7

Strong scalability 8

Weak scalability 9

Discussion 10

Introduction

The aim of this work was to provide an implementation of a 2d-tree made in parallel, both using *MPI* and *OpenMP*. A kd-tree is a common data structure, used for example in the KNN algorithm; for more details see [1].

In order to solve the problem, it is provided an hybrid code, which is discussed in the next sections, along with the performance analysis.

The code provided is written in C and compiled with GCC 9.3.0, OpenMPI 4.0.3 and OpenMP 4.5.

The performance analysis was obtained on ORFEO[1] cluster.

Algorithm

The first task faced to find the solution was to build the serial algorithm to construct the tree. Then the parallelism was added , which will be discussed in detail in the next sections.

The input of the algorithm is an array S of points $(x, y) \in \mathbb{R}^2$, with cardinality $|S| = n$. The data points are assumed to be homogeneously distributed in the space and the data set is also assumed to be immutable.

The proposed solution used a *divide and conquer* algorithm with two recursive calls. At each call, the splitting point is computed and added to the tree. At the same time, the data set S is partitioned into two sub-sets, one of larger elements of the split and one of smaller elements, which are passed to the recursive calls. The axis of split is chosen with a round-robin scheduling. The pseudo-code of the algorithm is:

Algorithm 1: Serial Kd-tree

```
   Input:  S is the input set of points
   Output: the tree $T$
1  T ← ∅
2  Function BuildTree($S$,$T$):
3      if $S$ is empty then
4          return
5      end
6      choose split axis
7      find $MAX$ and $min$ in $S$
8      $x_{split} \leftarrow \arg\min_{s \in S} |s - \frac{MAX-min}{2}|$
9      $S_{left} \leftarrow \{s \in S : s < x_{split}\}$
10     $S_{rigth} \leftarrow \{s \in S : s > x_{split}\}$
11     $T \leftarrow T \cup \{x_{split}\}$
12     BuildTree($S_{left}$,$T$)
13     BuildTree($S_{right}$,$T$)
14     return T
```

Where all the operation described in the algorithm 1 are computed respect the chosen axis.

The assumption of homogeneity of the points is used to select the splitting point as the closest element of S to the median of the chosen axis.

To partition the array S, with respect to the splitting axis (line 9 of algorithm 1), moving all elements smaller than x_{split} to the left and the largest elements to the right the following routine is used:

[1] https://www.areasciencepark.it/piattaforme-tecnologiche/data-center-orfeo/

Algorithm 2: Partitioning

```
Input: $S$, $\lambda$ such that $S[\lambda] = x_{split}$
Output: $S$ partitioned and the index of the splitting point
swap($S[n]$, $S[\lambda]$)
$i \leftarrow 0$
for $j \leftarrow 0$ to $j = n - 1$ do
    if $S[j] < S[n]$ then
        $i \leftarrow i + 1$
        swap($S[i]$, $S[j]$)
    end
end
swap($S[i + 1]$, $S[n]$)
return i+1
```

The computational complexity of finding the maximum and minimum, finding the splitting point and partitioning S are $O(n)$, so the complexity of the algorithm 1 is $O(n \log n)$.

Implementation

In this section it will discuss an hybrid (MPI and OMP) implementation of the algorithm already described. The main idea is to firstly divide the work among all the available MPI process and then build a tree using an OMP implementation of the algorithm 1.

Each MPI process build its tree using a data structure where a node is defined as follow :

```
struct tree_node {
struct data {
    float_t point[2];
} value;
int AxSplit;
struct tree_node *left;
struct tree_node *right;
}
```

The parallel routine that each MPI process use to build the tree is obtained simply put the recursive calls of the function *BuildTree* inside OMP tasks. The functions to find the max and min value and to partition the set S are left serial, even if it is provided an implementation that use OMP, because has been verified that it cause too much overhead and the performance deteriorates.

The real difficulties encountered during the construction of the program were founded trying to divide the work among the MPI processes and to re-build the entire tree at the end. The main issues were:

1. How send a tree through MPI_Send.

2. How assign to each MPI process the right part of set S and rebuild the tree.

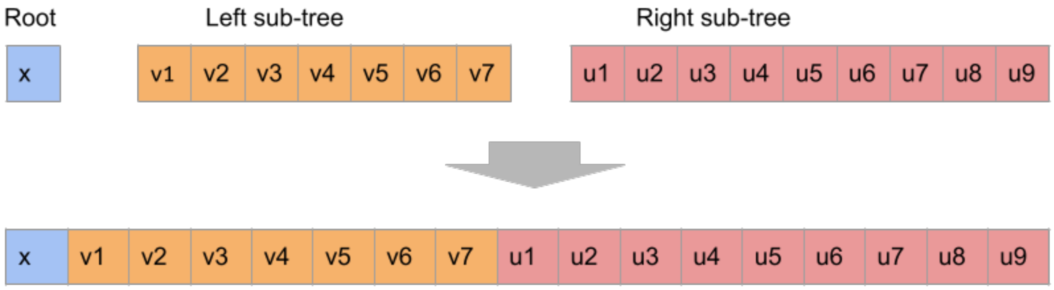


Figure 1: Merge two tree saved as array in one single tree.

Send a tree using MPI

In order to find a way to send a tree using MPI, the data structure used to save the tree was changed. This is due to the fact that MPI is meant to work better with arrays and continuous data in memory.

More specifically, it was build a function that convert the tree saved with pointers to struct to an array with the following rule:

- The tree is saved in-order in the array, so the first element is always the root.
- Each node of the array contain the indexes of its children.

Using arrays makes immediate to send the tree between two MPI process. Furthermore, given two tree v, u and a new data point x it is possible to build a tree in which x is the root and v is the left sub-tree and u the right sub-tree simply create a new array as is shown in figure 1. This useful propriety is used to merge all the trees built from different MPI processes in one single tree.

The conversion of the tree in an array is simply done traversing the tree.

Both convert the tree and merge two trees have complexity $O(n)$. In fact, also the merge function must go through the entire array to update the indexes of the children of each node. However, the merge function is implemented in parallel using OpenMP while the function that convert the tree in an array is serial.

Divide work and re-building the tree

A the start of the computation, only the master MPI process owns the set S. In order to divide the work, i.e. divide the set S among all the MPI processes, a hierarchical tree method is used as described in the figure 2 and in the algorithm 3.

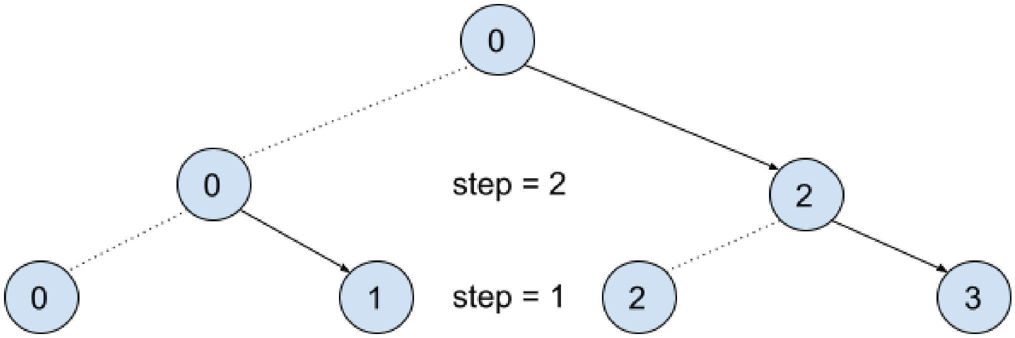


Figure 2: Example of divide work among 4 MPI processes

```
Algorithm 3: Divide work
Input: $P = \{\text{MPI processes}\}, |P| = M$
$step \leftarrow \max_i\{2^i : 2^{i+1} < M\}$
while $step \geq 1$ do
    if $p\%(2 \cdot step) == 0$ then
        if $p + step < M$ then
            $x_{split} \leftarrow$ FindSplitAndSort(S)
            send to $p + step$ the set $\{s \in S : s > x_{split}\} \subset S$
            $S \leftarrow \{s \in S : s < x_{split}\}$
        end
    end
    else if $p\%step == 0$ and $p \neq 0$ then
        $S \leftarrow$ the set received from $p - step$
    end
    $step \leftarrow \frac{step}{2}$
end
```

Basically, at each level of the tree in figure 2 the following steps are performed:

- The sending and receiving processes are calculated.
- Each sending process finds the splitting value of the kd-tree and partitions the set S with the functions described in the previous section (here it is used the parallel version).
- Each sending process send the right part of S and keep only the left part. The splitting value is saved.

The algorithm 3 is applicable with any number of MPI processes, however only using power of 2 is well balanced; in the other cases the data are not distributed equally. The number of MPI_Send and MPI_Recv is $M - 1$, where M is the total number of MPI processes. This is due to the fact that each sending processes has always itself as left child, so it makes only one MPI_send.

With the same method, applied in reverse, and using the tree saved as an array and its proprieties described above, the tree is rebuilt, as is shown in figure 3.

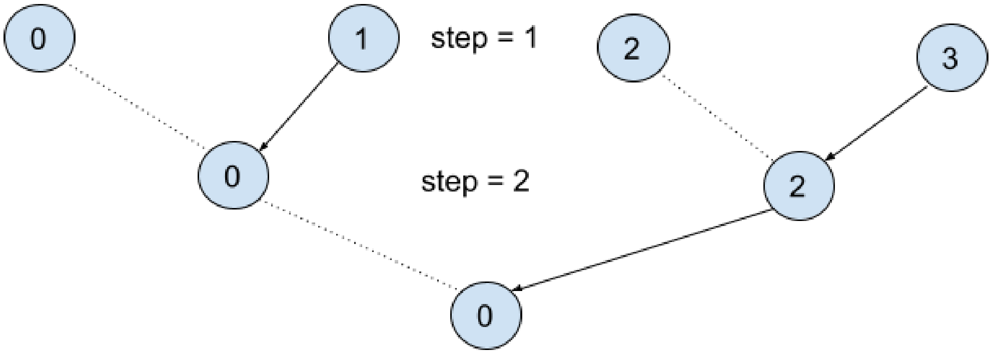


Figure 3: Example of merging the trees from 4 MPI processes

Full implementation

The complete implementation could be summarized in the following steps:

```
Algorithm 4: Implementation schema with $q$ MPI processes and $p$ threads.
Input: A set $S$ of point in $\mathbb{R}^2$
Output: A tree $T$, saved as an array
Divide the work from the master process among the $q$ MPI processes, saving the split index
  each time.
Build $q$ sub-tree, one for MPI process, using $p$ OMP threads.
Convert the sub-trees to arrays.
Merge all the $q$ sub-trees in one tree, saved in the master process.
```

Performance model and scaling

Since the nature of the implementation, it was difficult to build a mathematical model to describe the performance of the program in parallel. However, in the following paragraph is proposed an heuristic model experimentally verified.

Performance model

From the algorithm 4 follows that:

$$TotalTime(n,p,q) = SendingRebuilding(n,q) + BuildTree(n,p,q) + ConvertArray(n,q)$$

where n is the number of points, q the number of MPI processes and p the number of OMP threads for each MPI process.

Let be $T(n)$ the time of sorting and sending 1 message of size n. Since the master process will always send $log(q)$ messages:

$$Sending(n,q) = \sum_{i=1}^{log(q)} T\left(\frac{n}{2^{i-1}}\right) \approx T(n)\frac{2(q-1)}{q}$$

and since is experimentally verify that the sending part take (more or less) the same time of re-building the tree the time of the MPI part will be multiplied for 2:

$$SendingReciving(n,q) = 2 \cdot T(n)\frac{2(q-1)}{q}$$

The building tree routine carries out $log_2(n)$ recursive calls for each MPI process. So let be $\lambda(n)$ the time taken for one calls with size n:

$$BuildTree(n,q,p) = \lambda\left(\frac{n}{q}\right) + \frac{\sum_{i=1}^{log_2(\frac{n}{q})} 2^i \cdot \lambda\left(\frac{n}{2^i}\right)}{p}$$

The time taken to convert the tree in array is simply

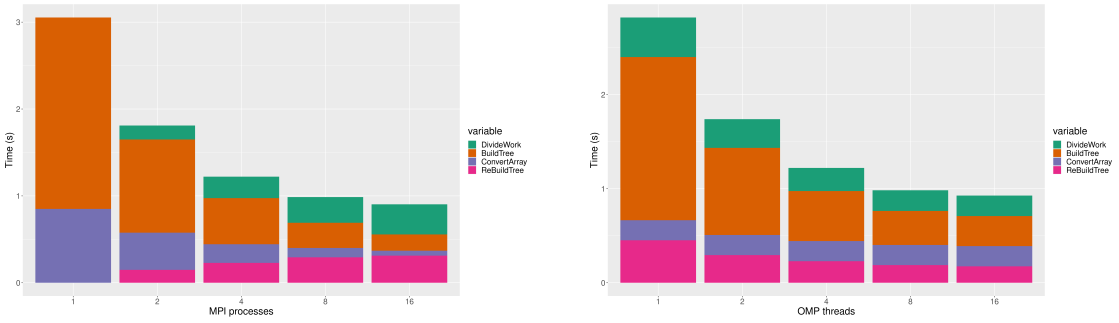
$$ConvertArray(n,q) = \frac{ConvertArray(n,1)}{q}$$

beacuse is always serial and with complexity $O(n)$.

Results

The program was run using up to 30 OMP threads and 24 MPI processes. The results presented are obtained on the GPU nodes. The program was execute mapping by *socket* both the MPI processes and the OMP threads.

In order to better investigate performance and scalability, the times of the different parts of the program were also measured. In particular, it was considered the time to send the set S among the MPI processes, the time to build the tree and to convert the tree to array and the time to re-build the tree. The figure 4 shows the impact of the various parts on the total time.



(a) 4 OMP threads

(b) 4 MPI processes

Figure 4: Time of the different part of the program with different numbers of MPI processes and OMP threads.

As expected, the time taken to divide the work and rebuild the tree becomes predominant as the number of MPI processes increase (Fig 4 (a)). In figure 4 (b) it is also possible to observe the impact of the parallelism in the function that find the splitting value and partitions the set S used in the distribution phase, as well as the function that merge the tree in the re-building phase. The figure 5 shows the total time with all combinations of MPI processes and OMP threads.

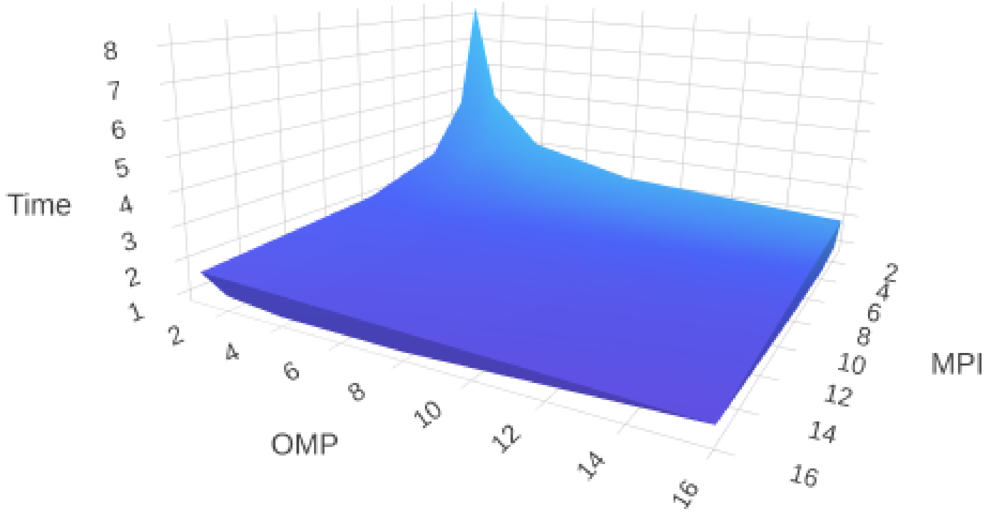


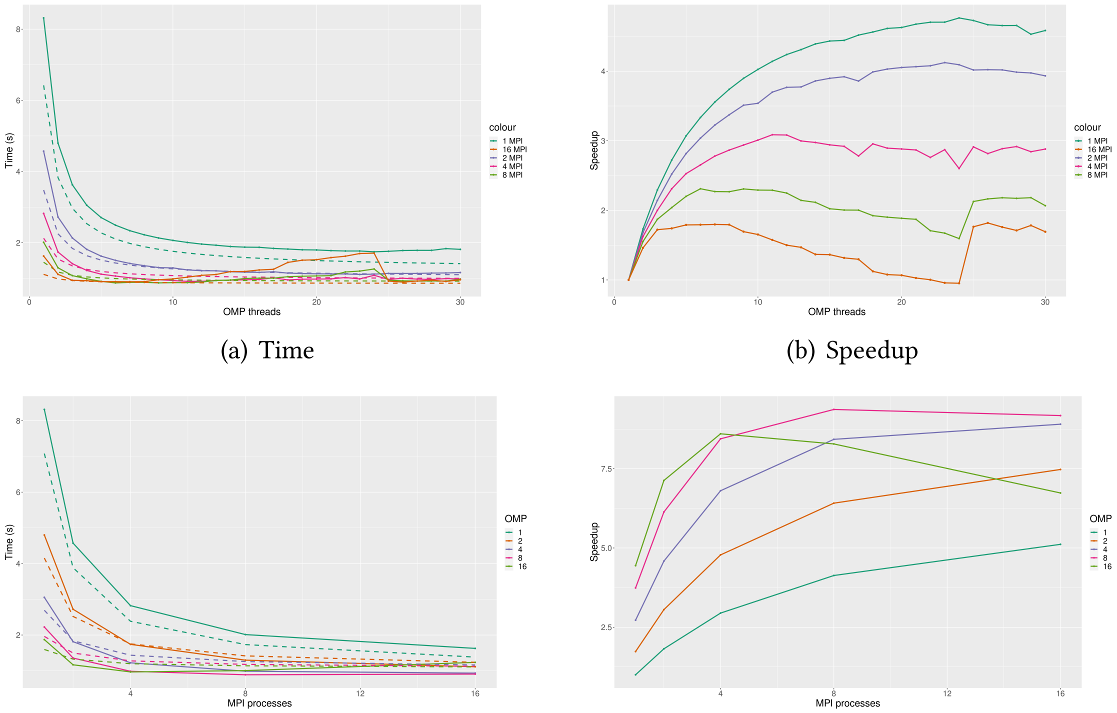
Figure 5: Total time with 10^7 points with different numbers of MPI processes and OMP threads.

Strong scalability

The strong scalability is investigated with $n = 10^7$ points. The figure 6 shows the time obtained with different number of MPI processes and OMP threads.

It is possible to notice that, as we expected from the Amdahl's law, after a certain point the gain given by the parallelism becomes stable. In addition, as the number of MPI processes increases, the speed-up decreases. There are two reasons for this behaviour:

- More MPI processes are used and more threads are working in the same physical core. In fact, there are 24 physical core available on one GPU node (48 with hyper-threading): if we want to map all threads in a physical core for each MPI process we must have $\textit{number of threads} \cdot \textit{number of MPI processes} \leq 24(48)$.
- More MPI processed are used and more time is taken to sending and receiving the messages.



(a) Time

(b) Speedup

(c) Time

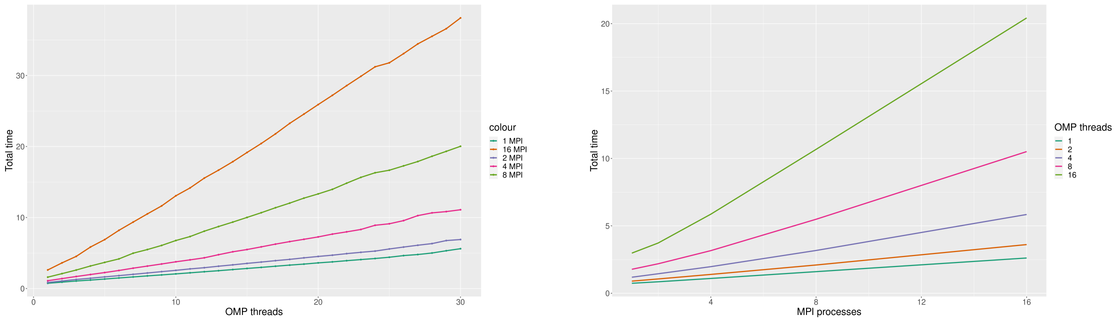
(d) Speedup

Figure 6: Total time depending from the number of threads and MPI process. The dashed lines are the expected value based on the theoretical model.

As we can see from the figure 6 (a) (b), there is a strange behaviour with 16 MPI processes. The time starts increasing until 24 threads, and then drops down. A possible explanation is that the time increases because of the overhead ($16 \cdot \#\text{OMP threads} \approx 100 - 300$ threads in the same node). However, it is still unclear why the time goes down with more threads, although it appears to be related to thread mapping: the same behaviour is observable in the THIN node between 12 and 13 thread, so when the thread count for each MPI process becomes greater than the number of CPUs in a socket.

Weak scalability

The weak scalability is obtained with 10^6 points for worker. The obtained results are shown in figure 7.



(a) (b)

Figure 7: Weak scalability for different numbers of MPI processes and OMP threads.

This results suggest that the MPI part is the one that limits most the weak scalability. This behaviour is not surprising : more MPI processes required more message passing with bigger dimensions. The time taken to divide the work, and rebuilding the tree, is summed up with the time taken for splitting and partitioning the set (which grows as well for the first splits) and produced the impoverishment of the performance.

Discussion

The problem assigned required more efforts than expected, especially for the MPI part. The main issue of the proposed solution is the time taken to distribute the work among the MPI processes and to re-built the whole tree.

From the performance analysis seems that the bigger limitation of the scalability is related to the fact that the parallelism increase as well as the tree construction increases, i.e. the first splits slow down the scalability.

During the construction of the algorithm it was choose to compute at each step the splitting point and partitioning the set, with a complexity cost of $O(n)$ at each step. Another solution could be to sorting the set at the begin for each dimension, and then selecting the splitting point as the element in the middle. However, this method have a initial cost of $O(n \log n)$ to sorting the data.

One small improvement of the program could be to parallelize the function that convert the tree in array: however this task may be tricky, since each position of the array is linked to the position of its child.

Instead of doing this, it might possible to build the tree directly with arrays: even if this solutions it is not so difficult to implement in a serial way, some complication arise when we want to parallelize, due to the fact that at each recursive calls must been calculated the right index in the array and the indexes of the child[2]. However, the time taken for the conversion is around the $9-10\%$ of the total time, so parallelize this part would not transmute into a great impact.

[2] In the GitHub page there is an implementation of the tree using only arrays. However, this solution was not delivered because it has poorest performance than the proposed one.

Another possible improvement could be not rebuild the tree at the end, implementing some routines to print or save the tree directly from each MPI process. This is maybe one of the fastest and simply solutions for this problem, however this really depend from the nature of the problem and from what we need to do with the tree.

There is also a false sharing problem that could be fixed when two threads are working in two part of the array with a distance smaller than 8 elements ($\frac{\text{cache line size}}{\text{point size}} = \frac{64}{8} = 8$), since all threads work in the same array. A possible solution could be to break the parallelism, or the construction of the tree, when the dimension S becomes too small.

Obviously, the last improvement that could be made is to focus more on optimizing each operation and write a more readable and efficient code.

References

[1] Martin Skrodzki. *The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time.* Mar. 2019.