Foundations of High Performance Computing

Assignment 2

Academic year: 2021-2022
Author: Erika Lena
Username: elena
Id: SM3500498

1 Premise

All the code, discussed below, used to build the kdtree has been developed in C using both MPI and OpenMP and applying an hybrid approach. The code has been compiled using GCC compiler versions 11.2.0, 9.3.0 and 8.4.0.
A simple Makefile is provided to compile the code; the code can be compiled both with and without the use of OpenMP.
A folder with the scripts used to measure performances is made available as well.
The code is available at the following GitHub page:
`https://github.com/erikalena/hpc_assignment2`

2 Introduction

The purpose of this project was to build a *kd*-tree data structure in parallel, using OpenMP and MPI tools.
A *kd*-tree is a binary tree, where each node is a k-dimensional point. Building the tree means to recursively split up the k-dimensional space through the use of hyperplanes. Each hyperplane is defined by a pair made of one k-dimensional point and an associated dimension, so that if p is the chosen point and y is the dimension currently selected, the corresponding hyperplane will be $y = y_p$. Essentially they are a special case of binary space partitioning trees.
In the figure below, we can see a visualization of resulting *kd*-tree on a very simple dataset made just 10 points.

$$D_1 = [[1, 2], [3, 4], [4, 1], [8, 3], [5, 6], [7, 8]]$$

```
$ cat final_kdtree.txt
(4.00,1.00)
    (1.00,2.00)
            (3.00,4.00)
    (5.00,6.00)
            (8.00,3.00)
            (7.00,8.00)
```

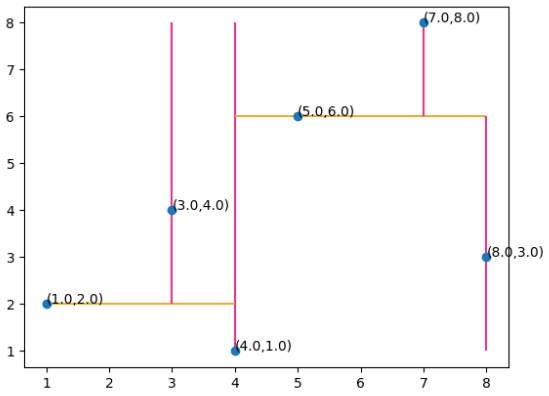


Figure 1: Textual and graphical visualization of *kd*-tree given dataset D_1

$D_2 = [[1, 2, 0], [3, 4, 2], [4, 1, 8], [8, 3, 5], [0, 9, 4], [10, 6, 3]]$

```
$ cat final_kdtree.txt
(4.00,1.00,8.00)
     (3.00,4.00,2.00)
          (1.00,2.00,0.00)
          (0.00,9.00,4.00)
     (10.00,6.00,3.00)
          (8.00,3.00,5.00)
```

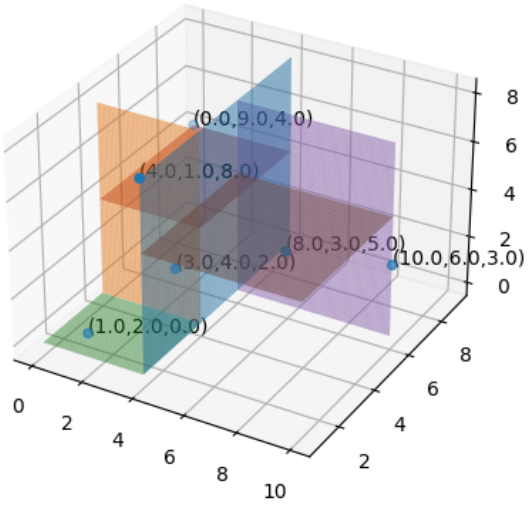


Figure 2: Textual and graphical visualization of *kd*-tree given dataset D_2

3 Algorithm

The approach used for building the *kd*-tree was to start by building it sequentially and then adding parallelism.
A recursive construction of the binary tree was performed as usual by calling the same function on left and right half-spaces at each iteration. The main thing to be established was in how to choose the splitting node. This choice was simplified by the following hypothesis:

- the data points are assumed to be homogeneously distributed in all the k dimensions;
- the data set is assumed immutable.

The dimension to be considered was chosen in a round robin fashion, a scan of the whole array was done to find maximum and minimum values and so to establish the median. Finally, the chosen *mid-point* was used to partition data in two halves, which will become left and right subtrees.

Algorithm 1 build_kdtree

```
// number of dimensions of the space
integer NDIM

Node Build-kdtree(data_t[ ] points, integer n, integer axis)
 if n == 0 then
 |  node = null
else
 |  // decide new splitting axis
 |  new_axis ← (axis+1)%NDIM
 |
 |  // find splitting point
 |  mid ← partitioning(points, n, new_axis)
 |   n_left ← mid
 |   n_right ← n - n_left -1
 |   node.split_point ← points[n_left]
 |   node.axis ← new_axis
 |
 |  // recursively build-up left and right subtrees
 |  node.left ← build_kdtree(points, n_left, new_axis)
 |
 |  node.right ← build_kdtree(points+n_left+1, n_right, new_axis)
 └
```

Each node is made of a structure which contains information about which is the associated splitting axis, the corresponding splitting point and references to the left and right subsets determined by the division of the space.
The most relevant part to be discussed is in how to partition the data points, which is done be applying the idea behind *quicksort* algorithm, choosing as pivot the median value with respect to a given axis.

4 Implementation

The *kd*-tree was implemented as a simple binary tree, using pointers to left and right sub-trees. Each node represents an hyperplane splitting a sub-region of the space; nodes are simple structs made of an integer, which is the splitting dimension, and three pointers, one to the data point and the other two to the next left and right nodes.
Each data point is in turn a struct made of an array of k floats (or doubles, depending on the required accuracy).

The most relevant part was in deciding how to parallelize tree construction and how to combine OpenMP and MPI. An hybrid approach was used, which allows to see how each of the possible *processes-threads* combination behaves.
To each MPI process was assigned a portion of our data so that each of them could construct its subtree in parallel with the others; while, OpenMP was used to parallelize the instructions to build each subtree, using *tasks* in particular so that left and right branches construction could proceed in parallel.

Then, a key point was in how to assign data to MPI processes. This was pretty difficult, since just one process had to read all the data in order to have an idea of our domain; then, we can start splitting data points and sending one halves to another process which will do the same on its turn and so on, until all the available MPI processes are up ad running.
Each process gets its own data and starts constructing its own subtree but, since this subtree is just a subregion of a larger domain, I wanted the subsets to be assigned following a precise order such that it was possible to "easily" reconstruct the whole tree. For this reason, I managed to assigned data to processes so that it could be possible to read the whole tree and in particular, to read it by a depth-first search (DFS).

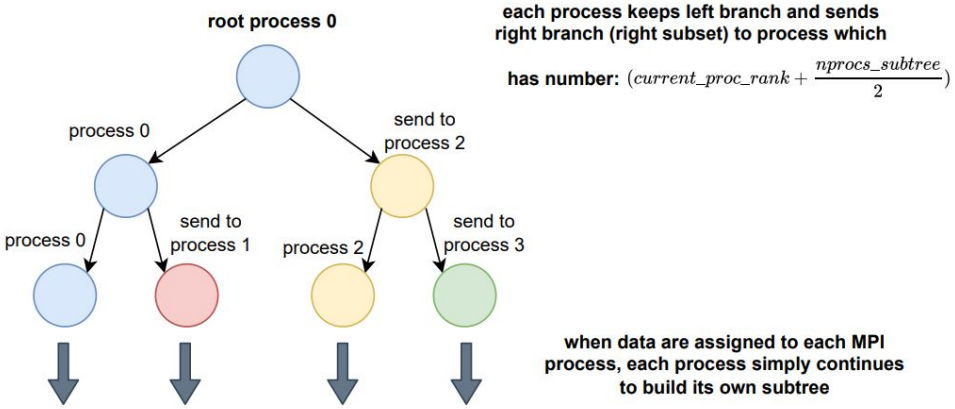


Figure 3: The figure show how data are partitioned between 4 MPI processes.

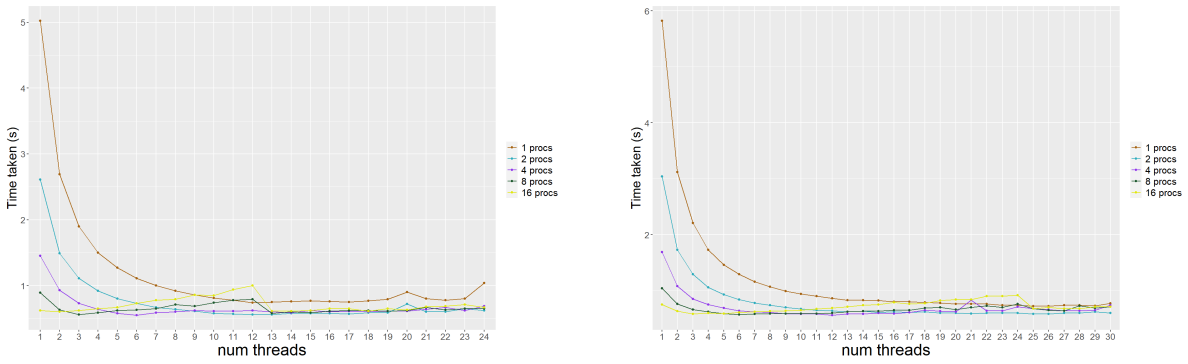
This is not the handiest way to assign data to processes, but it was nice to orchestrate the processes to make them work like this.

To facilitate tree construction while exploiting parallelism on left and right branches, MPI processes were considered to be a power of two, and so if otherwise, the program aborts.

5 Performance modeling and scaling

5.1 Modeling

The code has been run on Orfeo [1] platform, both on CPUs and GPUs node; the interesting thing was in comparing different performances obtained by using OpenMP, MPI and a combination of both of them.
Results are shown with respect to following choices: MPI processes are mapped by **socket**, as well as **OMP_PLACES** is set to **sockets**.
So, proceeding by steps we look at how our code behaves when increasing the number of threads. When OpenMP is used, the construction of the tree is parallelized, in particular we have that each thread works on a given subtree, while the operations needed to divide the data in branches are not parallelized further since it resulted in a deterioration in performances, probably due to threads coordination and overhead. Nonetheless, data partitioning is already "indirectly parallelized", since data are splitted at each level and operations on the two halves are performed in parallel by different threads.



(a) CPU node

(b) GPU node

Figure 4: Measured times on CPUs and GPUs node obtained by varying the number of OpenMP threads with respect to 1,2,4,8 and 16 MPI processes; problem size is 10^7 data points.

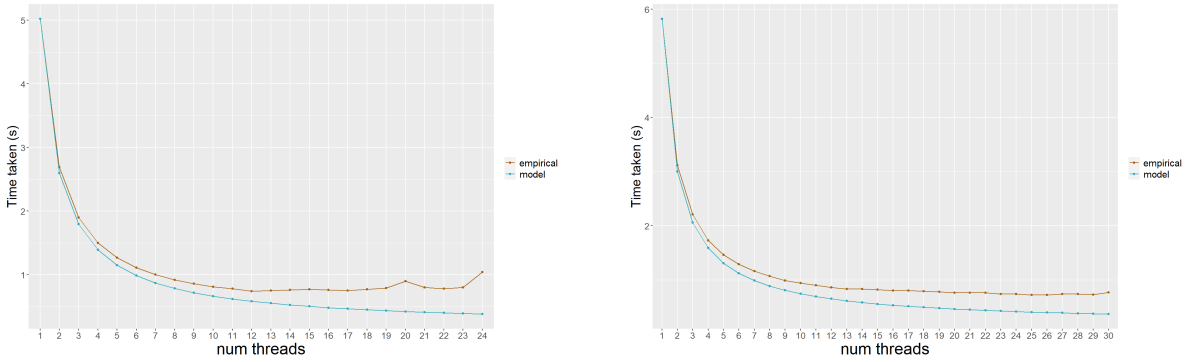
The only part which remains sequential is the first call to the function which partitions the data by finding the maximum and the minimum, the median and then swapping data like in *quicksort* partitioning part; so in the end, they are basically three readings of all the data points.
From what has just been said, we can define a very simple and trivial model, which measures

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

the time needed to run in parallel as follows:

$$T_p(N,p) = \frac{f_p}{p} + T'_s + \frac{T_s - T'_s}{p}$$

where T_p is the parallel time on a problem of size N using p workers, f_p is the parallel fraction of code, in our case the construction of subtree, T'_s is the sequential time needed to read all the data, while T_s is the time needed for recursively partitioning data at each level (this part is not parallelized), which is also the most expensive part of the algorithm.



(a) CPU node (b) GPU node

Figure 5: Results obtained by using just one MPI process and varying the number of threads; the plots compare empirically measured times on CPUs and GPUs against the simple model provided.

Now, we look at how times change with respect to the number of MPI processes used keeping fixed the number of threads.

Even if the time needed to exchange data at the very begin is consistent, once the data are splitted the time required for each MPI process to build its subtree decreases. This is not only because the workload is smaller, but also because concurrent memory accesses are fewer.
Another very simple model is provided in order to see how parallelization should work; like before, we considered what happens with just one thread varying the number of MPI processes to simplify. So, the fraction of code relative to subtree construction should be $\frac{f_p}{p}$, since data are splitted and sent to each MPI process before starting to build subtree, but we have to add the time needed to send data at the very begin, so the proposed model is:

$$T_p(N,p) = \frac{f_p}{p} + T_s\left(\frac{N}{p}\right) + T_{send}(p)$$

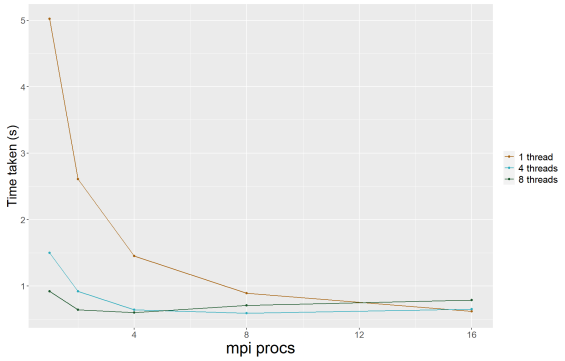
where T_p is the parallel time on a problem of size N using p workers (here p MPI processes), f_p is the parallel fraction of code, in our case the construction of each subtree, p is the number of MPI processes, T_s is the time needed to recursively partition all the data for each of the MPI processes, while T_{send} is the estimated time to exchange data between

processes using message passing, this time accounts also for the the first k splits performed before each of the MPI processes starts working on its subtree.

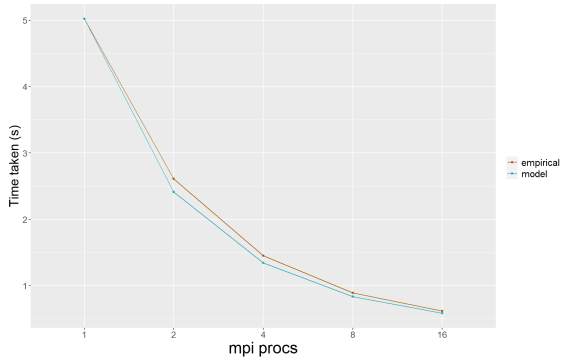
The main difference, is in how we can estimate the time needed by *partitioning* part T_s. In the previous model we divided the time for the number of threads, after having subtracted the first sequential reading of the whole data structure; this was of course an approximation because not all the threads start working at the same time.
Now we can provide a better estimation of T_s, since the time needed for first k=$log(p)$ partitions is already taken into the considerations when measuring the whole time needed by the *MPI part*, that is to say it is included in T_{send}.
Once each process is working, the time T_s needed for recursively partitioning, which is also the heaviest part, can be estimated as:

$$T_s(n) = 3 \cdot t_s \cdot log(n)$$

where t_s is the sequential time needed to read the data structure of size n, while $log(n)$ is the number of times the partitioning functions is called, the depth of the subtree.



(a) Empirical times obtained by using just one thread varying the number of MPI processes.



(b) Comparison of empirically measured times against the simple model provided varying the number of MPI processes.

Figure 6: Results obtained varying the number of MPI processes used on CPU nodes, using 1 thread and a problem size of 10^7 data points.

Finally an attempt to model results obtained when OpenMP is used along with MPI is made, since the algorithm used seems to be faster when we use a suitable combination of threads and MPI processes. In particular best results are obtained with 4 or 8 MPI processes and a small number of threads.
A very simple model is proposed by a small variation from the previous one, when subsets of points have already been sent to each MPI process, we add parallelism by considering also threads in the construction of each subtree:

$$T_p(N,p) = \frac{f_p}{p_M \cdot p_O} + \frac{T_s\left(\frac{N}{p_M}\right)}{p_O} + T_{send}(p)$$

where p_M is the number of MPI processes and p_O is the number of available threads. However, this is a very simple model which is not able to capture the complexity given also by interaction of MPI and OpenMP used together and most importantly it does not consider we do not have the physical hardware to run all this threads together. That is to say, the model considers that all the threads are available for each MPI process to build its subtree, but of course this is not the case.

Considering 4 MPI processes and mapping them by socket, means we are running 2 of them on each socket. Using 6 threads means each process is able to spawn 6 threads and so 12 different threads can run concurrently on the socket saturating it. We can observe the same behaviour when 8 MPI processes are used with number of threads set to 3. From 6 threads on, the timings remain more or less constant and we do not obtain the increase in the performances that we would expect.

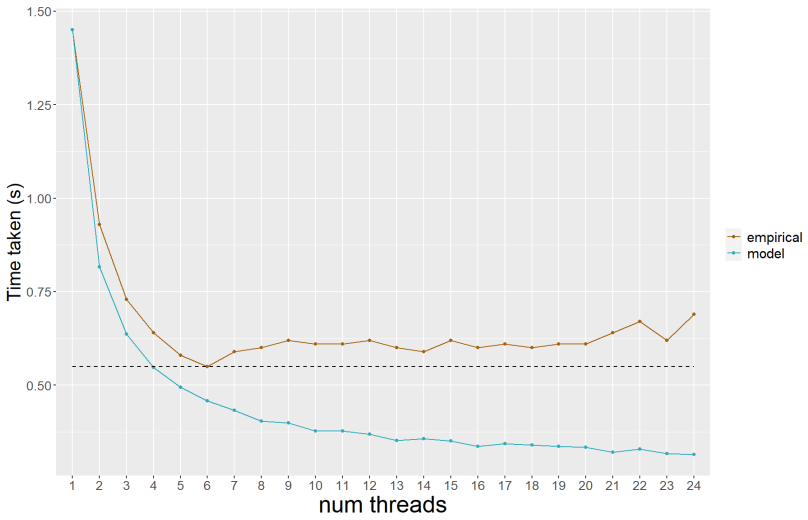


Figure 7: Timings obtained, varying the number of threads keeping fixed the number of MPI processes to 4 and problem size to 10^7, compared to expected ones.

Finally, in the plot below the measured times are shown with respect to all the possible combinations of threads and MPI processes.

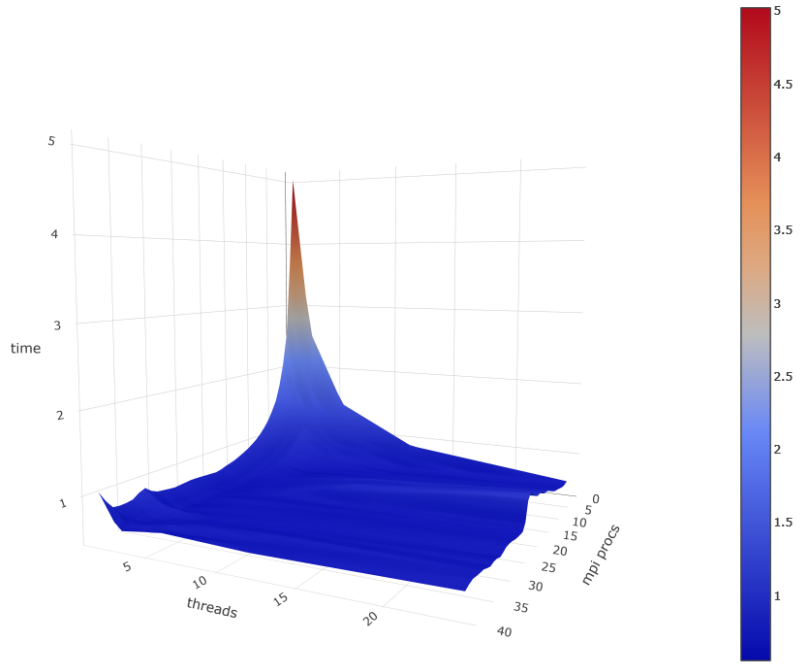
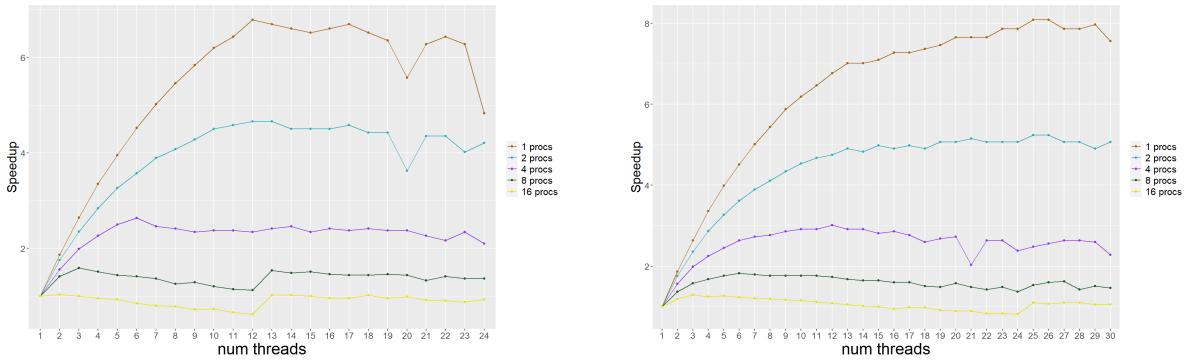


Figure 8: Timings obtained on a CPU node keeping problem size fixed to 10^7

5.2 Scaling

5.2.1 Strong scaling

We have already seen how times change with respect to the number of MPI processes and the number of threads, while keeping the global size of the problem equal to 10^7. Here we observe which is the speedup in our execution time. The speedup is in general given by the ratio between the serial time and the parallel time: $S(N, p) = \dfrac{T_s}{T_p}$.
In the figure below the speedup obtained by increasing the number of threads is plotted. We can observe the usual trend provided by the Amdahl's law.



(a) CPU node

(b) GPU node

Figure 9: Estimated speedup on CPUs and GPUs node, varying the number of threads, problem size equal to 10^7 data points.

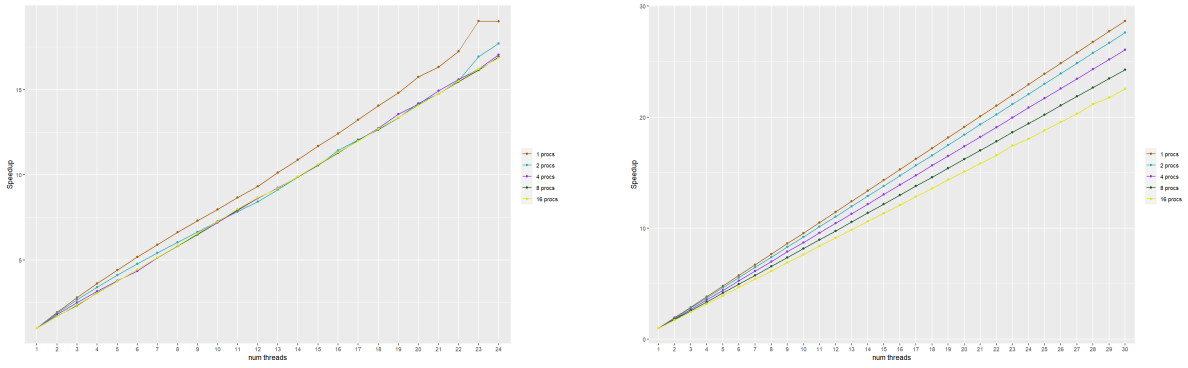
It is quite difficult to estimate the upperbound defined by the Amdahl's law, since the only part of our code which can be considered to be intrinsically sequential (f) is the first reading of the array, but this would result in a huge upperbound for the speedup given by $\frac{1}{f}$, however the not sequential code can not be completely parallelized, as we can not make all the *processes* start working together.

5.2.2 Weak scaling

Weak scalability is investigated as well, since when we increase the size of our problem, usually the parallelizable part increases more than the sequential one. So, by keeping fixed the workload for each *process* and increasing the global size of the problem, the speedup should now be linear in number of *processes*, as stated by Gustafon's law. Speedup is computed as:

$$Sp = p + f \cdot (1 - p)$$

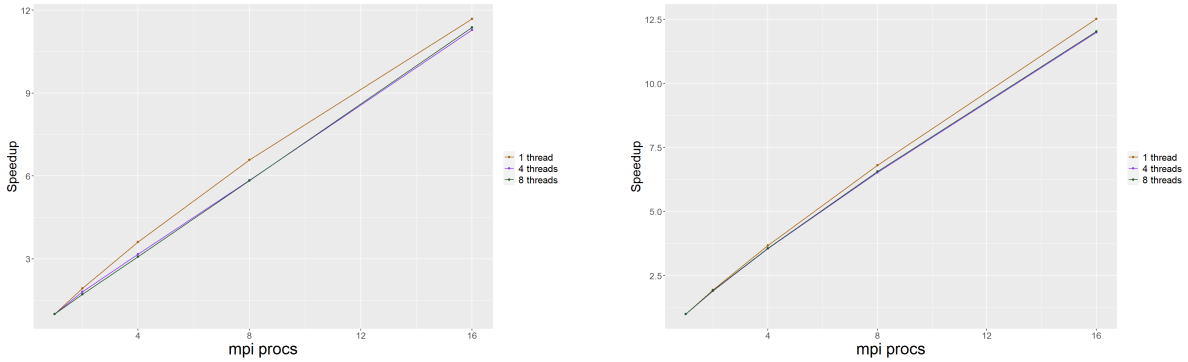
where f is the fraction of intrinsically sequential code.



(a) CPU node (b) GPU node

Figure 10: Estimated speedup on CPUs and GPUs node varying the number of threads, and keeping fixed the workload assigned to each thread to be equal to 10^6 data points.

The same holds if we look at the speedup with respect to the number of MPI processes used, keeping the number of threads fixed.



(a) CPU node (b) GPU node

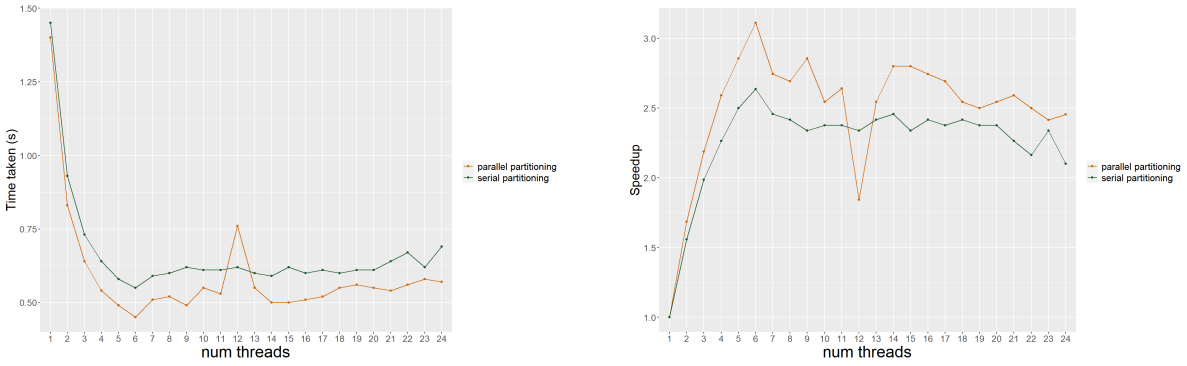
Figure 11: Estimated speedup on CPUs and GPUs node varying the number of MPI processes, and keeping fixed the workload assigned to each MPI process to be equal to 10^6 data points.

6 Discussion

The main part of the algorithm and the most expensive one is data partitioning. Everything is in deciding how to split the data and in moving data to left and right *sub-branches*. Here we adopt a simple strategy which consists in finding at each call the maximum and the minimum, then the median point and finally move points in our data structure with respect to this *pivot*. All of this requires 3 readings of the whole data structure (N) and it is recursevely called $log(N)$ times, so its computational cost is $O(Nlog(N))$.

To improve the research of *mid-point* and so the data partitioning task, one viable way was to further parallelize it; anyway this approach appeared to be slower, probably due to

waste of time in coordinating the threads; while the main gain was given by the fact that problem size is halved at each iteration. I have tried to implement parallelization on *data partitioning*, by using OpenMP, and I have observed that timings increased.
To overcome issues related to the number of threads, another possibility was to increase parallelism just in the *MPI part*, that is to sat on the first $log(p)$ *splits* (where p is the number of MPI processes), when not all the workers have been assigned their portion of data yet.
The only drawback is that we have to repeat code which basically do the same thing, partitioning data, keeping two versions of it: one to be called for the first $log(p)$ *splits* and the other for subsequent calls.
Results obtained using this approach, are shown in figure below, we can see there is a slight improvement in performances, which is interesting at least until the number of threads spawn by each process is quite limited ($<$ number of cores on one socket). When 4 MPI processes are used, 2 MPI processes at level 1 (as it is show in figure 3) are calling the partitioning in parallel and so they are asking for a certain number of threads. Since we have 2 sockets, each MPI process works on one sockets and asks for t threads, the cores available on each sockets is 12 which coincides with the a local maximum in timings. This happens if we set **OMP_PLACES** equal to **sockets**, while if we ask for threads to be placed by core, we do not see this jump, but timings are constantly higher. This is due to the fact that we decided at the very begin to map MPI processes by socket, so it seems reasonable to adopt the same policy for threads, in order to keep threads close to the corresponding MPI process (thread 0). Results obtained using *socket-socket* policy are almost identical to those obtained using *core-core* policy.



(a) CPU node (b) GPU node

Figure 12: Comparison between results obtained with and without parallelism on first k *splits* of our data points. Times are compared running on 4 MPI processes and varying the number of threads; problem size is 10^7 data points.

Another viable way to handle data partitioning was to order all the data at the begin, keeping a data structure which maintains the values of data points ordered for each dimension and for all but the first dimensions the indexes of the data point in the array ordered with respect to first dimension. In this way, we do not need to scan all the points each time in order to find the maximum, the minimum and the median, since data points are already ordered, and we do not even have to swap and partition each time our data points. However, this approach has an initial cost of $O(Nlog(N))$, given by the an ordinary sorting algorithm like *quicksort*; the sorting has to be applied for each dimension in order to avoid the scan of the data structure to find the *mid-point*. A potential drawback of this technique would be if data are not homogeneously distributed along all

the dimensions, as in this case the array should be scanned even if already sorted.

The possibility of not homogeneously distributed data was taken into account and a function which chooses the axis with respect to data extension has been provided, even if this requires a further scan of the data structure.
Another possible pitfall related to the initial data is the possibility of overlapping points, even if this possibility is quite remote since we are dealing with *float* or *double* data types.
To conclude, a brief parenthesis about the choice to use **float** or **double** type: of course doubles allow us to increase the precision in number representation, but this comes with a quite significant increase in time, in particular the time needed to exchange data between MPI processes is more than double the one obtained by using single precision floating point numbers.