Assignment 2: High Performance Computing

Matteo Boi (s225241)

March 21, 2022

GitHub Repository: https://github.com/BoiMat/HPC.git

1 Introduction

Aim of the assignment is to construct a kd-tree, a data structure use to represent a set of k-dimensional data in order to make them efficiently searchable. The implementation of the tree is done in serial and parallel, both using the OpenMPI and OpenMP libraries. The final algorithms are then presented, with their strengths and weaknesses and an analysis on the strong and weak scaling of both implementations is drawn studying the results obtained on both Thin and GPU nodes on ORFEO. At last, some problems and possible solutions are discussed.

1.1 KD-Tree

Kd-trees were first introduced by [3], and remain now days a good choice to perform a whole series of k-nearest neighbor related operations. Since it easier to visualize, we can focus on the 2-dimensional case, that is the one we will work with.

In a finite point set P each point has two values that are important, its x- and y-coordinate. The basic idea is that we start at the root, splitting P with a line along one of the axis in two subsets of roughly equal size. The splitting point is stored at the root. The left subset P_l is stored on the left subtree and P_r to the right one. The two subsets are then split again with a line along the other dimension and the points on each side of it are stored on the left and right subtrees of the children. Each child stores the splitting point. The procedure is repeated alternating the splitting dimension until there are no more points to split (Fig. 1). A more formal and detailed description can be found in [1], [2], [4].

2 Algorithm

Given the basic concept we can focus on the actual implementations used for this work. The dataset is composed of N 2-dimensional floating points in the range $[0, 1]$. The choice of the data type is customizable through a compiler flag between *float* and *double type* and the actual values are assigned via a pseudo-random numbers generating function that should ensure the uniform distribution of the points. The size of the dataset can be given as input to the program and is set by default to 10^8 points.

Tree nodes & Utility Functions

Algorithm 1 Node struct definition

typedef struct kd-node {
    $split \leftarrow$ *k-point*
    $^*left \leftarrow$ *pointer to left child*
    $^*right \leftarrow$ *pointer to right child*
    $axis \leftarrow$ *current splitting dimension*
}

A node of the tree is defined as in Algorithm 1, with a 2-d point representing the splitting point, two pointers pointing to the left and right child respectively and an integer storing the axis of split.

Each implementation of the code comprises some utility functions, in particular one to generate the array, one to swap two points of the array and one to choose the splitting point. Since the first two are trivial, we will just spend some words on how the splitting point is chosen.

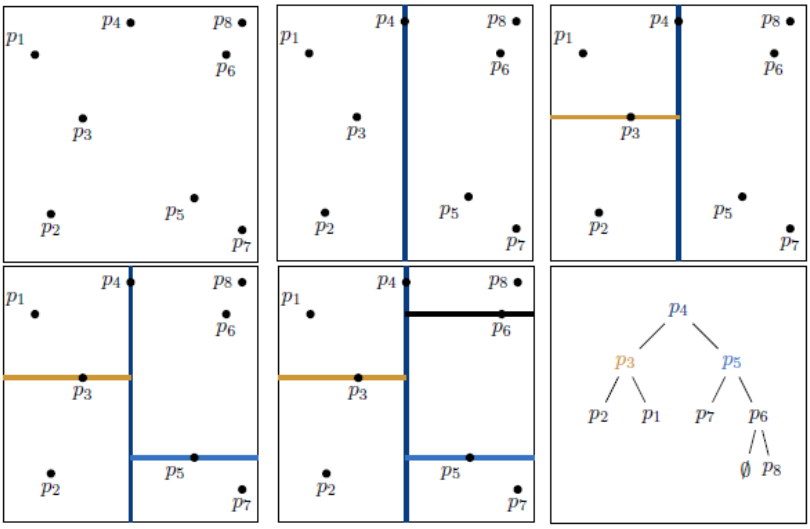

Figure 1: 2d-tree building procedure on eight points. In the last figure the complete tree is shown. Figure from [4].

Given the assumption that the data is homogeneously distributed along every dimension we decided to use the median of the array as splitting point. This requires that the array is somehow ordered along the current dimension. Actually a perfect order is not required, it is sufficient that points lower than the median are on the left side and greater on the right and this is achieved thanks to a basic implementation of a *quickselect algorithm* inside the function.

As mentioned three implementations are presented.

2.1 Serial

Algorithm 2 Building kd-tree function

Require: a point set P, size N, splitting dimension $axis$, number of dimensions $ndim$.

$^*Node \leftarrow$ *pointer to current node initialized*
if N $==$ 0 **then**
  return NULL;
end if
if N $==$ 1 **then**
  $Node.split \leftarrow P$
  $^*Node.left \leftarrow NULL$
  $^*Node.right \leftarrow NULL$
  return
end if

$n \leftarrow find\_splitting\_point(P_1, P_N, axis)$

$Node.split \leftarrow n$
$Node.axis \leftarrow axis$

$axis \leftarrow (axis + 1)\%ndim$
$P_l \leftarrow \{p_i \in P | p^{axis} \leq q^{axis}, p_i \neq q\}$
$P_r \leftarrow \{p_i \in P | p^{axis} \geq q^{axis}, p_i \neq q\}$

$^*Node.left \leftarrow build\_kd\text{-}tree(P_l, N, axis, ndim)$
$^*Node.right \leftarrow build\_kd\text{-}tree(P_r, N, axis, ndim)$
return $^*Node$

The main function of the code is the *build_kd-tree()* function which pseudo-code is shown in Algorithm 2. It is a recursive function that takes as input the pointer to the first element of the array, the size, the axis along which

we want to start the split and the dimensions of the points. The first step is to check if the size is lower or equal to 1. If it is equal to one the function returns a node with the only point present as splitting point and $NULL$ pointers to the left and right child. If this is not the case the algorithm proceeds finding the median through the *choose_splitting_point()* function and storing it in the current node. Then, left and right child are found by recursively calling the *build_kd-tree()* function for the left and right side of the array with the split represented by the median. After all the nested calls of the function have returned, a pointer to the root node is returned and the programs ends.

In the next two sections we will describe the differences between the serial and a parallel implementation of the code both in the case of *MPI* and *OpenMP*.

2.2 MPI

Algorithm 3 Building kd-tree function with MPI

Require: ... , number of MPI processes $size$, current $depth$.

if N == 0 ...
if N == 1 ...

if $2^{depth} \geq size$ **then**
    As the serial case..
else
    $recv \leftarrow 2^{depth} + myrank$ (receiver process)
    $N_r \leftarrow size\ of\ P_r$
    $info \leftarrow \{N_r, axis\}$

    $MPI\_Send(info, recv)$ ← sending required information
    $MPI\_Send(P_r, recv)$ ← sending right part of the array

    $^*Node.left \leftarrow build\_kd\text{-}tree(P_l, N, axis, ndim)$
end if
return $^*Node$

The main problem in a parallel framework is how to divide the work, without creating load imbalance and making too many unnecessary communications between the processes. This could create some serious overhead that would make the program not scale properly. Here we have two options, the easier one is to give the work to the master rank while the other wait to receive a part of the array. The master will start the building process, splitting the initial array until the numbers of subarrays is equal to the number of active processes. At this point it sends one to each process. Then, each one can start building its subtree. The problem with this configuration is the load imbalance, since the master process will do the majority of the work, having to deal with the full array and the initial splits.

The implementation we decided to adopt should, at least in part, solve this problem. The master process finds the first splitting point but then immediately sends the right subarray to one of the free processes, keeping the left subarray for itself. Once a process receives its part of the array can start building the tree, sending the right subarray if there are still free processes or building also the right part if there are not. This greatly reduces the amount of work assigned to the master process keeping the same number of communications calls between processes. Unfortunately we do not solve the load problem completely, since the first processes in the chain receive bigger subarrays and have to work more than the others. In addition, the last processes have to wait in receive mode for a long time before they can start working.

Fortunately, the fact that we are dealing with trees makes it easy to calculate from whom each process will receive the data using its rank and the total number of processes. The only additional information each process needs to send is the number of points that it will be sending and the current axis of split.

We show in Algorithm 3 the pseudo-code, with highlighted in red the additions with respect to the serial implementation.

2.3 OpenMP

Switching from MPI to OpenMP means passing from completely independent memory regions to a common shared space. This relaxes some aspects of the problem but raises new ones. Since there is no communication needed the code is really similar to the serial one, although there are some differences. Even if it is not taken into account when

Algorithm 4 Building kd-tree function with OpenMP

...

$axis \leftarrow (axis + 1)\%ndim$
$P_l \leftarrow \{p_i \in P | p^{axis} \leq q^{axis}, p_i \neq q\}$
$P_r \leftarrow \{p_i \in P | p^{axis} \geq q^{axis}, p_i \neq q\}$

$k \leftarrow cache_line_size/sizeof(2d\text{-}point)$
if $N > k$ **then**
    #pragma omp task
    $^*Node.left \leftarrow build\_kd\text{-}tree(P_l, N, axis, ndim)$
    #pragma omp task
    $^*Node.right \leftarrow build\_kd\text{-}tree(P_r, N, axis, ndim)$
else
    $^*Node.left \leftarrow build\_kd\text{-}tree(P_l, N, axis, ndim)$
    $^*Node.right \leftarrow build\_kd\text{-}tree(P_r, N, axis, ndim)$
end if
return $^*Node$

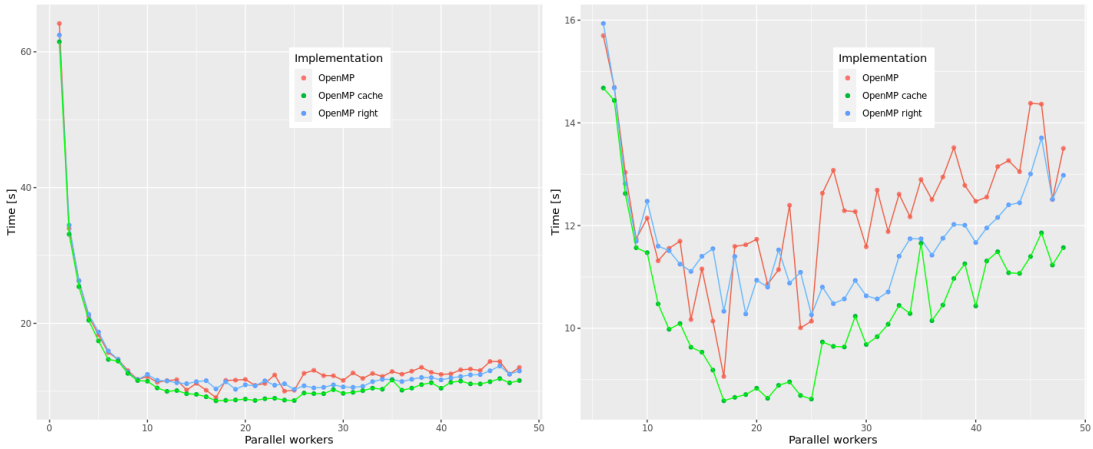

Figure 2: Comparison in the runtime for the different OpenMP strategies in a strong scaling scenario.

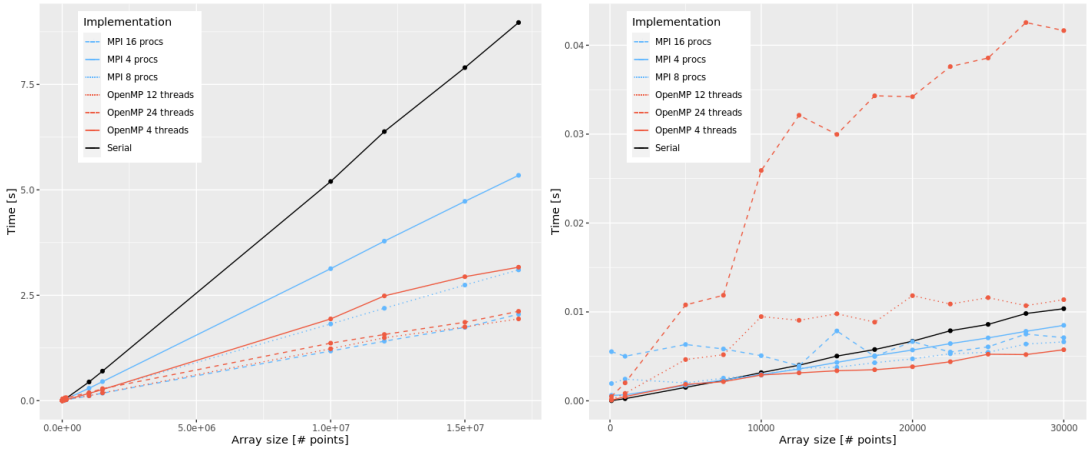

Figure 3: Comparison of the timings between the serial and the parallel cases for different MPI processes and OpenMP threads varying the array size.

timing the code, the generation of the array is conducted in parallel. Then, a parallel region is initiated and the master process starts building the tree. When it gets to the point where the build function is recursively called to build the left and right child, we have two possible choices. The first is to assign the construction of both children to a single task while the second is to create two different tasks for the left and right child respectively. The advantages in the first case are that a smaller number of tasks is generated and need to be managed, and in addition we can leave the current node shared, since only one thread will write data on it (namely the pointers to the children). However, this strategy suffers from a huge problem of load imbalance in the first phases of the construction, given the fact that the first threads have to deal with a really big array, constructing both children while the others wait. This problem is at least alleviated using the second strategy, i.e. one task constructs the left child and one the right **[a]**. This anyway has some flaws, like the increase in the number of tasks generated, especially when we get near the leafs, where the subarrays are really small. Both strategies returned similar results in terms of time. Taking the example of MPI a third strategy was implemented: when a thread reaches the point of constructing the children, creates a task only for the right one and it keeps constructing the left part **[b]**. This strategy was designed to reduce the overhead present when a thread finishes a task and have to take a new one. The last problem we tried to solve is when the subarray becomes really small. In fact, after a certain depth the overhead of completing a task, wait and get a new one, coping the variables and so on becomes greater than the actual computation that follows. To solve this problem we added a condition that limits the generation of threads when N becomes "small enough". We chose as threshold the moment when the number of points of the subarray fits in a cache line. Finally in Fig. 2 we show the results for strategy **[a]**, **[b]** and **[a]** plus the task limiting condition in a strong scaling scenario. Last strategy resulted to be the best one, as it is clearly visible in the zoomed image on the left.

3 Scaling

3.1 Orfeo

Results are obtained using Orfeo computational nodes. Thin nodes comprise Intel Xeon Gold 6126 CPU 2.60GHz with two sockets, 12 cores per socket, one thread per core. The L1 (32 kB) and L2 cache (1 Mb) are core private while the L3 (19 Mb) is socket shared.

GPU nodes have an Intel Xeon Gold 6226 CPU @ 2.70GHz with again two sockets, 12 cores per socket, but two threads per core. The L1 (32 kB) and L2 cache (1 Mb) are shared between threads in the same core while the L3 (19 Mb) is socket shared.

3.2 Serial vs Parallel

The first question we want to answer is: does parallelism actually speed up computations? In fact, if the work that needs to be done is not "enough" for each process or thread, the overhead caused by setting up the parallel environment, spawning threads and communication in case of MPI processes can lead to performances that are worse than the serial case. Therefore the problem is characterized by two factors that we can modify to study the behaviour: the amount of work, represented by the size of the array and the parallel overhead related to the number of OpenMP threads or MPI processes. To investigate this we varied the size of the array between 10^2 and 10^8 points, and then ran

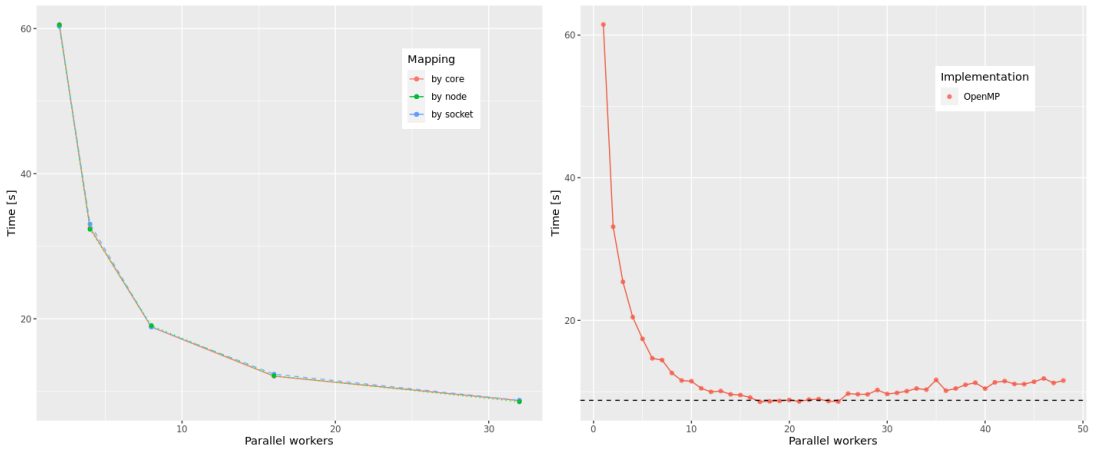

Figure 4: Runtime in a strong scaling scenario for the MPI (left) and OpenMP kd-tree (right) on Thin nodes on Orfeo.

the code with 4, 8, 16 MPI processes and 4, 12 and 24 OpenMP threads. We show the results in Fig. 3. As we can see from the left figure, there is no doubt that for a large number of points the parallel implementations perform much better than the serial one. If however we zoom in in the range between 100 and 30000 points we observe a totally different behaviour. As expected, the overhead present in the parallel cases makes the use of a parallel framework not worth it. In the case of MPI, for a low number of points the communication between processes causes an increase in the runtime proportional to the number of communications (= number of processes).

The OpenMP case is even more interesting, in fact whenever a parallel region or loop is initiated or stopped, there is some negligible overhead. In addition, tasking constructs require some nontrivial computation or bookkeeping in order to figure out which thread computes the next task. In the case of 24 OpenMP threads, these factors cause the runtime to be four times the serial one, while it does not impact too much for small numbers of threads that on the contrary will not scale too well for large numbers of points.

3.3 Strong Scaling

In a strong scaling scenario the amount of work stays constant (10^8 points in our case) while the number of parallel workers varies. The goal in this case is to minimizing the time to solution. For what concerns MPI we are forced by the implementation to use only 2^k processes. Given the architecture (described in Sec. 3.1) the tests were conducted with 2, 4, 8, 16 processes on the same node and 32 on two different thin nodes. Instead we don't have any limitations on the number of OpenMP threads that can be spawned in each parallel region, and tests were conducted using from 2 to 48 threads. This however does not result automatically in better performances as we will see now. In Fig. 4 we show the results obtained for the two implementations.

In the case of MPI given the small amount of communications the overhead is negligible with respect to the computational time, therefore even with different mappings of the processes there is not a significant difference in the runtime. In contrast a different behaviour is observed in the OpenMP case. When the number of threads exceeds the number of physical cores (24 in a thin node) the time increases again, this is also visible in Fig. 2. There are many sources of overhead connected to this: the difficulties in managing a large number of threads and the tasking constructs, keeping track of which threads are doing what, scheduling and cache coherency. Anyway there is no real advantage on having more threads than physical cores.

This analysis can be carried on studying the behaviour using GPU nodes instead of Thin nodes. Given the active hyperthreading, the available cores are in this case 48 per nodes instead of 24. In Fig. 5, where the timings for both MPI and OpenMP are plotted, we can see that the behaviour described above in the OpenMP case is not present anymore, and the runtime reaches a plateau around 12 threads.

In Fig. 6 we finally show the results obtained in terms of *speedup*, calculated as the ratio between the runtime of the serial program $T(1)$ divided by the parallel one $T(N)$:

$$Speedup = T(1)/T(N) \tag{1}$$

with N the number of parallel workers for both Thin and GPU nodes. The OpenMP implementation scales better than the MPI one, at least until the number of OpenMP threads exceeds the number of physical ones, where we have a drop in performance. For the MPI case, unfortunately given the hardware architecture and the power-of-two processes limitation we cannot observe what the behaviour is with more than 32 cores. We can hypothesize that the speedup will reach a plateau and stop scaling. In the case of GPU nodes the OpenMP speedup is not limited to the number

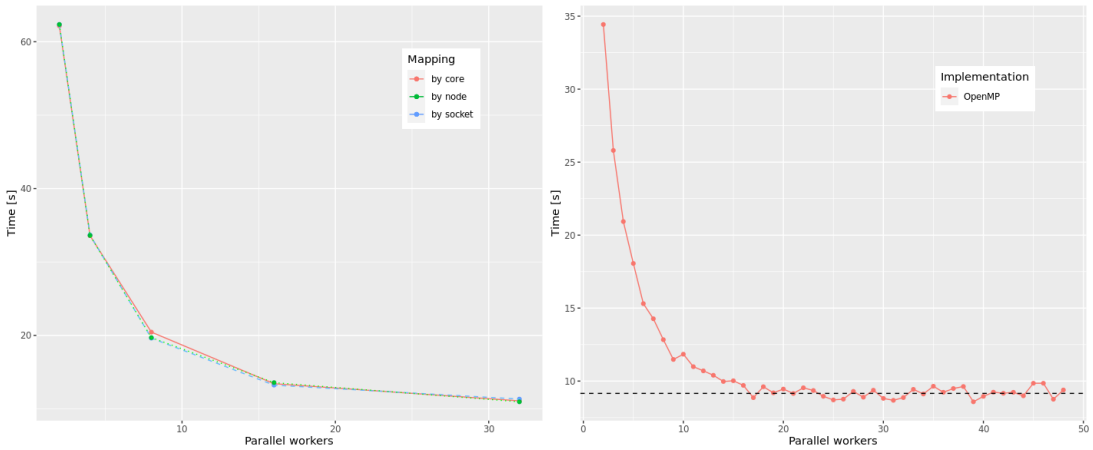

Figure 5: Runtime in a strong scaling scenario for the MPI (left) and OpenMP kd-tree (right) on GPU nodes on Orfeo.

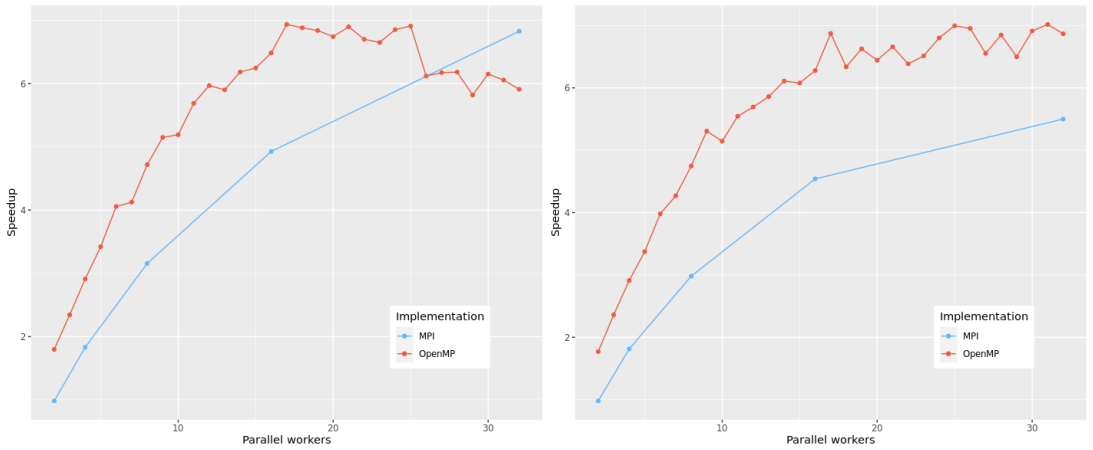

Figure 6: Performances in terms of strong scaling for the MPI and OpenMP kd-tree on both Thin (left) and GPU nodes (right) on Orfeo.

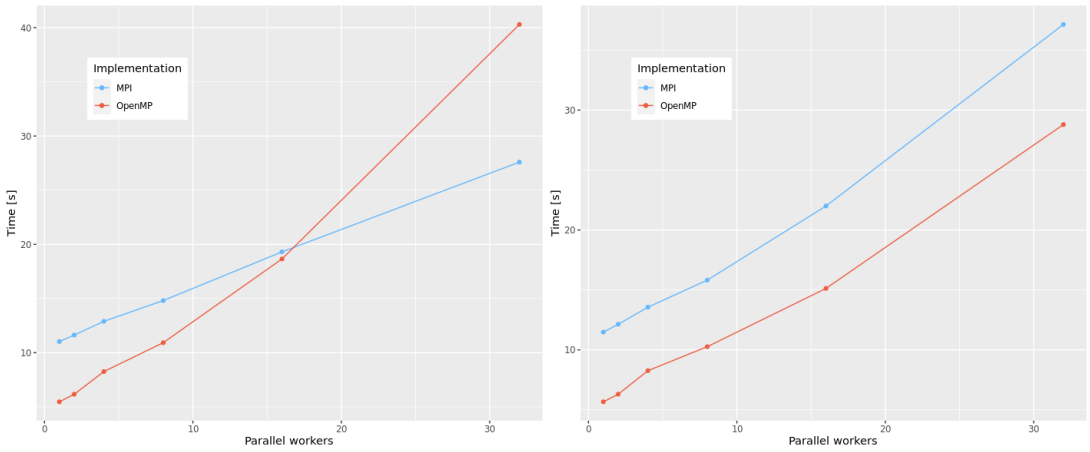

Figure 7: Runtime in a strong scaling scenario for the MPI and OpenMP kd-tree on Thin (right) and GPU nodes (left) on Orfeo.

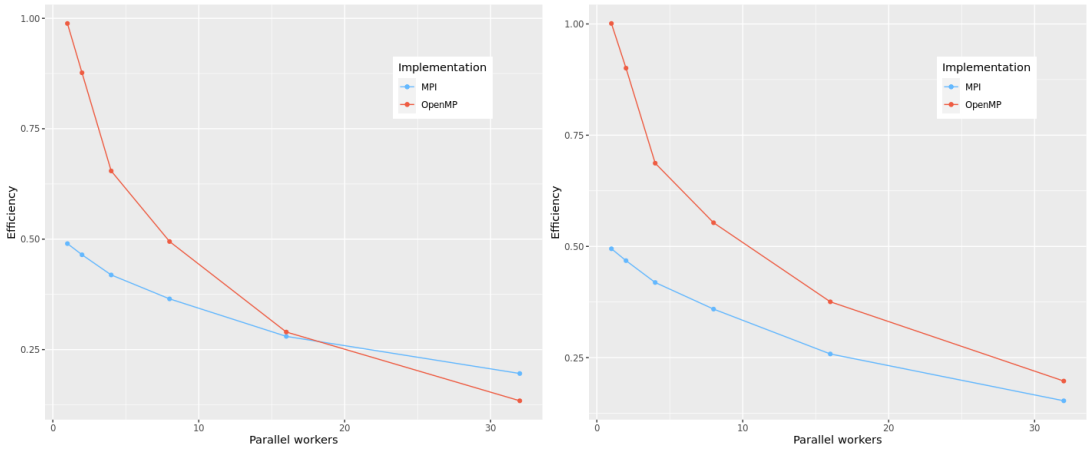

Figure 8: Performances in terms of strong scaling for the MPI and OpenMP kd-tree on both Thin (left) and GPU nodes (right) on Orfeo.

of cores and outperforms the MPI one. The gap between the two implementation is substantial. Part of the problem could be due to the communication time between MPI processes, given the fact that in the first iteration the master process has to send half of the array ($5 * 10^7$ points) to the receiver process and so on but some more deep problems may be present.

3.4 Weak Scaling

In the case of weak scaling, both the problem size and the number of parallel workers vary. This results in the ideal case in a constant workload per parallel worker and time to solution. In order to evaluate our model in this framework, we measured the timings using 1, 2, 4, 8, 16 and 32 parallel workers and scaling the array size linearly such that $size = N * 10^7$, with N the number of parallel workers. In Fig. 7 we show the results in terms of runtime for the MPI and OpenMP implementation on both Thin and GPU nodes while in Fig. 8 the results in term of weak scaling efficiency, calculated as:

$$Efficiency = T(1)/T(N) \tag{2}$$

where $T(1)$ is the amount of time to complete a work unit with one worker and $T(N)$ the amount of time to complete N of the same work units with N parallel workers. In the idea case the efficiency is represented as a constant horizontal line in $y = 1$. As it possible to see from graph both implementations do not weakly scale. The main reason is probably given by the already mentioned load imbalance present especially in the first iterations of the programs.

4 Discussion & Conclusions

Even if we presented many interesting results, there are many ways to improve the presented implementations. First of all starting from the serial implementation of the code. The choice of using a quickselect algorithm to partially sort the array performs better with respect of a quicksort one, but we are still not taking into account the types of points we are dealing with. A better choice would be to create an array of pointers pointing to the elements of the array and perform the quickselect on it instead of the actual data. In this way instead of moving 2-dimensional double floating points (16 bytes) we would just move 8 bytes pointers. We have also to keep in mind that if the number of dimensions increases, the size of the points will increase too, while the pointers' size stays the same. Another point is the choice of the splitting dimension. Again here it works thanks to the assumption of uniform distribution of the points, but it could be improved for a more general case by finding at each depth level the dimension where the points have the bigger spread and the splitting would be performed on that dimension.

For what concerns MPI the main problem is that the tree is split in subtrees on different processors. To reconstruct the tree there are many difficulties: first the tree cannot be built in a linked list fashion anymore, with pointers connecting a node and its children, given the fact that pointers in a specific memory region cannot live in another. A solution is to build the nodes in an array-based representation and send them to the master process that will reconstruct the tree. However here we are adding not only additional communication costs but also the reconstruction time. A better solution could be to make each process find only the splitting point for each iteration to save some communication time, and then let the master process use them to reconstruct the tree. Lastly, the power of two processes limitation should be resolved. Unfortunately for lack of time, this implementations are not covered here.

In the case of OpenMP few more improvements could be made, first of all the sorting of the array (especially in the early iterations) could be done in parallel to decrease the initial load imbalance. This is not a trivial task especially if one wants to avoid load imbalance. Second, the use of tasks makes all the variables used *firstprivate* and copied in the thread' stack while this is not necessary and some could be set to shared. Another observation concerns the choice of blocking the task generation when the subarray fits in the cache line (≤ 8 points in our case). This could be a suboptimal solution and a better one could be found with some tuning of the parameter.

At last some better results may be achieved using some compiler optimization flags, namely `-o1, -o2, -o3`.

References

[1] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 12 1972.

[2] M.T. Berg, M.J. Kreveld, and M.H. Overmars. *Computational Geometry: Algorithms and Applications*. 01 2008.

[3] Jerome Friedman, Jon Bentley, and Raphael Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 09 1977.

[4] Martin Skrodzki. The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time. 03 2019.