FOUNDATIONS OF HPC
Second assignment

Second report
Master in High Performance Computing
Universitá degli studi di Trieste
Trieste, Italy

presented by:
Alexander Cristhoper Trujillo Ochoa

February 25, 2022

Chapter 1

KD-Tree : Parallel implementation

1.1 Introduction

For this assignment we implemented a KD-Tree data structure, focusing in the specific case where K=2. About this structure first it is defined as a collection of nodes which can point to a left or right child node, and the ones without child nodes are called leaves. Each node will contain an splitting point which we select as the median of a sorted multidimensional array along one of its dimensions. To select the splitting dimension for every node we used a round-robin fashion selection through them, which it is enough considering that the data is homogeneously sparse.

The importance of this data structure falls in the idea of optimizing problems that requires each element to check every element, for example collision checking, K-nearest neighbors, etc. Normally a naive implementations for them will have N squared complexity, but if we consider the construction of a KDtree (construction of complexity NlogN), once obtained this structure, those algorithms could use it and be more efficient, this means now the complexity will be NlogN, even though for each new data set you will need to construct a new KDtree the time optimization is totally worth it.

1.2 Algorithm

In order to parallelize this algorithm we used a Hybrid (MPI+ OpenMP) approach.
To start we divided the total tree into two parts, one which we call the master tree and its depth will be determined by the number of available MPI processors (MPI depth) , this tree will contain only the first nodes and branches of the whole tree considering the total array until the total number of nodes in the last depth level is equal to the number of processors. This is because as the first nodes needs more points to sort, we use all the available threads to to complete those. This part implementation is shown in the Fig 1.1
The second part we call it the independent child forest, once depleted the possibility to use various processors to obtain one single node, each MPI process will start to construct its own tree independently using OMP threads this is shown in Fig 1.2 and 1.3

Each process takes care of one side of the tree, depending of the number of processors we can take care of more branches as we can increase the depth size of the tree (number of levels below the head node) and start the parallelization. So for this reason we consider the number of processors always a power of 2 and each process will use the same original data creating its own independent KDtree according with the corresponding branch. To obtain the splitting points, for each depth level we call the merge sorting algorithm which will sort along the chosen dimension only the portion of the data corresponding to each node, the splitting dimension is obtained using a round robin fashion of the previous dimension chosen.
At the end of the algorithm each process will return a pointer to the head node where only contains its own tree.

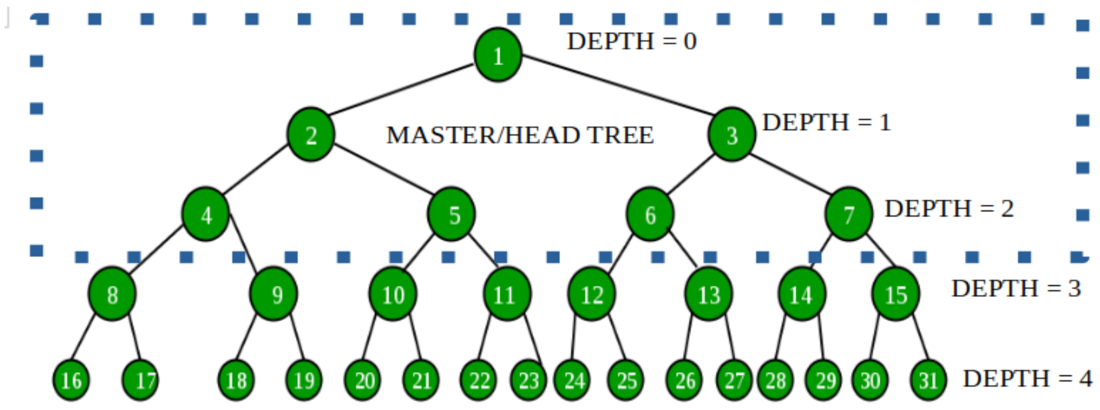

Figure 1.1: Graphic explanation of the master tree construction, for this case we have 4 MPI processes, so the depth of the master tree will be 2

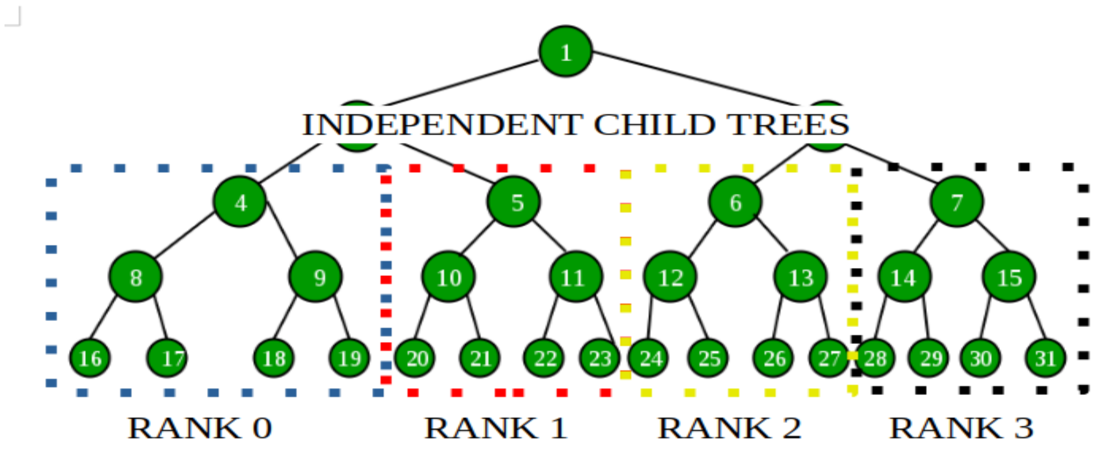

Figure 1.2: Construction of the independent child trees, each MPI process will create its own tree, considering in total 31 points, the final depth of the KDtree will be 4

For this Hybrid implementation we will take advantage of the recursion approach in the tree construction algorithm and we used OpenMP to parallelize it by generating tasks considering the divide and conquer approach. As well we use OMP threads to deploy one extra level of parallelism once there is one single MPI processes on the child tree.

We will implement a flag array which contains binary digits depending of the OMP rank so in this way, each thread will follow a unique path in the child tree according to these flags, constructing the nodes and omitting certain branches that will be constructed by other threads.

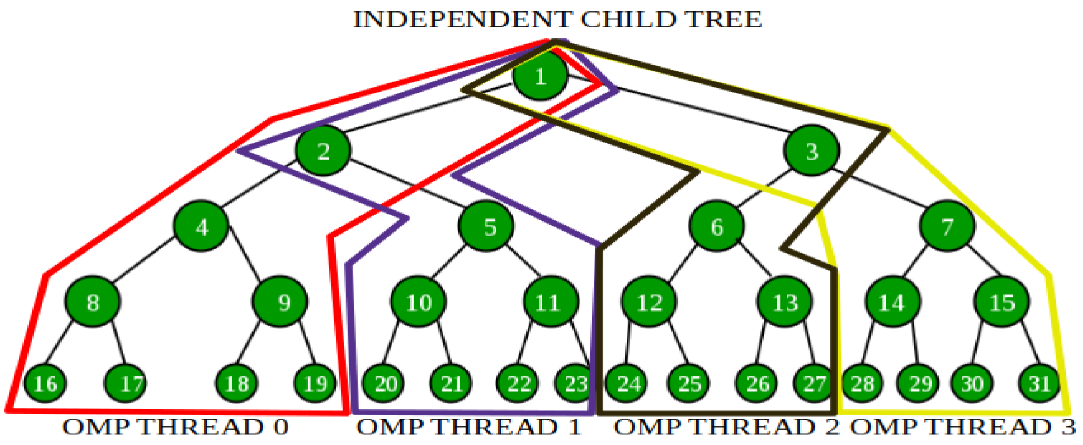

Figure 1.3: Each child tree is built using OMP threads by using flags which indicates a unique direction per thread, in this way the whole tree is traversed

1.3 Implementation

1.3.1 MPI staging

First we created a data structure for each node, which contains a pointer to a left , right node, splitting point and dimension. MPI processes read an external file containing all the points, this is stored in a array of KDpoints, also we allocated this array only one time for the parallel sorting (To use only one malloc and avoid excessive allocation overhead). Also we call a function getdepth (similar to log 2) which will be obtained considering the number of processes available at the start of the program, as we said we consider it always a power of 2.

```
struct kdnode {
 int split_axis;
 kpoint splitting_point;
 kdnode *left, *right;
}
depth = get_depth(size);
```

```
kpoint *array
readArray(array,size_array);
```

Now each MPI process proceeds to create its master tree so the threads will aid in the process to obtain the splitting point, in the construction of the master KDtree we used OpenMP directives to create tasks for each division (this tree creation uses divide and conquer approach) also we can set a certain threshold (TASKSIZE), this is to avoid excessively granularity and overhead of tasks creation which diminishes the global performance of the algorithm. Once the tasks are created the dormant threads starts to work on the tree creation.

```
//Creation of the master_tree
#pragma omp parallel
 {
 // Only one single thread will spawn other threads
 #pragma omp single nowait
 head_tree = build_master_tree(array, start, end, axis,
  ↪  depth);}
```

```
build_master_tree(points, start, end, axis, depth)
{
 int N= end - start+ 1;
 if(!depth){
  return NULL;}

  int median = split_point(points, start, end);
  node->point= points[median];

  if(depth){
   depth--;
   #pragma omp task
   node->left = build_master_tree(points, start,
    ↪  separation_index-1, axis, depth);

   #pragma omp task
   node->right = build_master_tree(points, separation_index+1,
    ↪  end, axis, depth);

   #pragma omp taskwait
   }
  return node;
}
```

Now as the master tree has been built we separate the total array in the total number of MPI processes and each of them will start to create a KDTree independently.

1.3.2 Linking OpenMP with MPI

As each MPI processor takes care of a determined tree, more branches will be created by means of OMP threads, an flag array filled with 0 and 1 according to its rank will point an unique path. Each 0 will indicate descend to the left node and 1 descend to the right node, in this way each rank will descend to construct a independent partial construction of the tree as shown in figure 1.3.

```
kdnode* binary_tree;
local_start=rank*size_array/size +1;
local_end=(rank+1)*size_array/size- 1;

#pragma omp parallel
 binary_tree =  build_kdtree(array, local_start, local_end,
  ↪  flags);
```

```
build_kdtree(points, start, end, flags){
 if(flag){
  node->right=NULL;
  node->left = build_kdtree( points, start,middle, flags+1);
  return node;}
 if(!flag){
  node->left=NULL;
  node->right = build_kdtree( points, middle,end, flags+1);
  return node;}
}
```

Once each thread gets the level point where there is no overlapping with other threads a independent KDtree will be constructed from there recursively: Starting with the base case where if the number of data points is 1 or 2 we create one or two nodes containing the remaining points and the node pointers are set to NULL, then we return the node.
If that is not the case we call the function splitting point which need as arguments the pointer to the global array, the start and the end indexes which marks the bounds of the partial sorting. This function will return the index of the median considering the new partially sorted array, in this case quickselect was used.
Next the function splitting dimension returns the index of the dimension for the next sorting operation using a round robin fashion.

```
build_kdtree(points, start, end, axis)
{
int N= end - start+ 1;
if( N == 1 ){
 node->left= NULL; node->right= NULL;
 node->split_point = *(points+start);
 return node;}

 //This partially sorts the array moving numbers lesser
 //than median to the left, and greater to the right
 quick_select(array, start, end, median)
 median = splitting_point(points, N, start , axis);
 splitting_dimension(axis);

 node->split_axis= axis;
 node->split_point= points[median];

 node->left = build_kdtree(points, start, median-1, axis);
 node->right = build_kdtree(points, median+1, end, axis);
 return node;}
```

This implementation follows the distributed data approach where each processor will have a child KDtree and the main processor will have the master tree, the last part of the implementation es finally link the master tree with the child trees.

At last, we have utilities to measure the walltime spent on the KD tree creation, flag encoder and functions where each rank can print its own part of the tree.

1.4 Performance modeling and scaling

1.4.1 Algorithm complexity

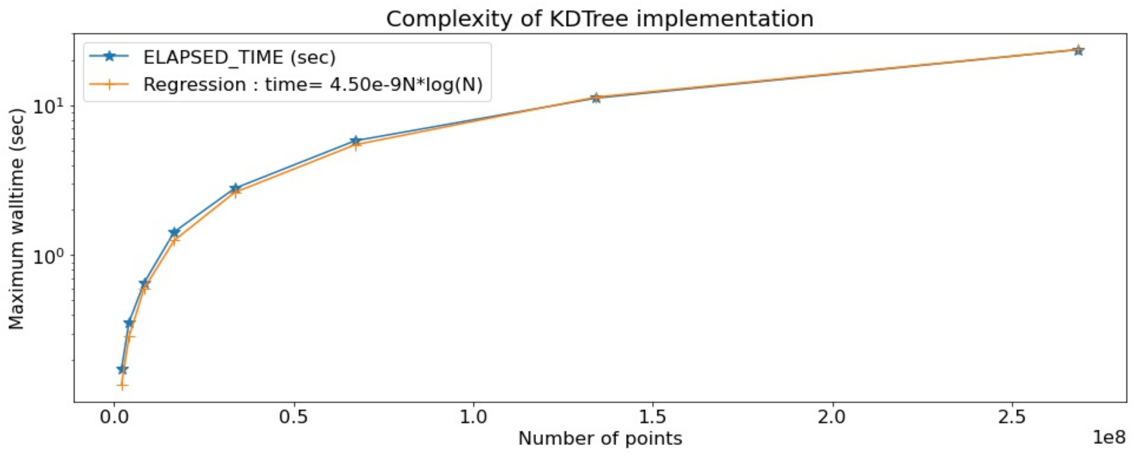

Figure 1.4: NlogN behaviour of the KDTree implementation, for this run 16 MPI processors were used

It is in fact well known that the complexity of the construction of a KDtree is k*nlog(n) in big O notation , where k is the number of dimensions and n the number of points. According to the Fig. 1.4, if we fit a curve of this style we obtain the following parameters.

1.4.2 Scalability behaviour using Hybrid approach

For this section we ran 2 cases, the strong and weak scalability, but we need to select the correct correlation between MPI and OMP, we fixed the number of points and MPI processors, the variable here will be the number of OMP threads. It is worth to mention that the MPI binding is by sockets and we got the following results in the table 1.1.

Points	# MPI Processors	# OMP Threads /MPI proc	Time
10^8	8	1	11.00590
10^8	8	2	8.56053
10^8	8	4	9.08749
10^8	8	8	9.84644

Table 1.1: OMP thread scaling behaviour

We observe that for more than 4 threads per MPI process this will saturate the time performance, so for the next part we will fix 2 OMP threads associated with each MPI processor to check the strong scalability by increasing the number of MPI processors.

1.4.3 Strong Scalability

This scalability depends purely on the number of processors and the fraction of the serial part of the algorithm according to Amdahl's law[3], so we obtained the following data shown in table 1.2 and the plot in the Fig 1.5

Points	# MPI Processors	# OMP Threads /MPI proc	Time
10^8	1	2	36.92265
10^8	2	2	19.97264
10^8	4	2	12.33233
10^8	8	2	8.42391
10^8	16	2	6.73822

Table 1.2: Strong scalability data

1.4.4 Weak Scalability

Now if we consider that the parallel section increases with the amount of work, this means that the greater our resources is better to use it for heavier works, this is established by Gustafson's law, so we obtained the following data shown in table 1.3 and the plot in the Fig 1.5.

Points	# MPI Processors	# OMP Threads /MPI proc	Time
4194304	1	2	1.73232
8388608	2	2	1.75374
16777216	4	2	2.24199
33554432	8	2	2.97237
67108864	16	2	4.67617

Table 1.3: Weak scalability data

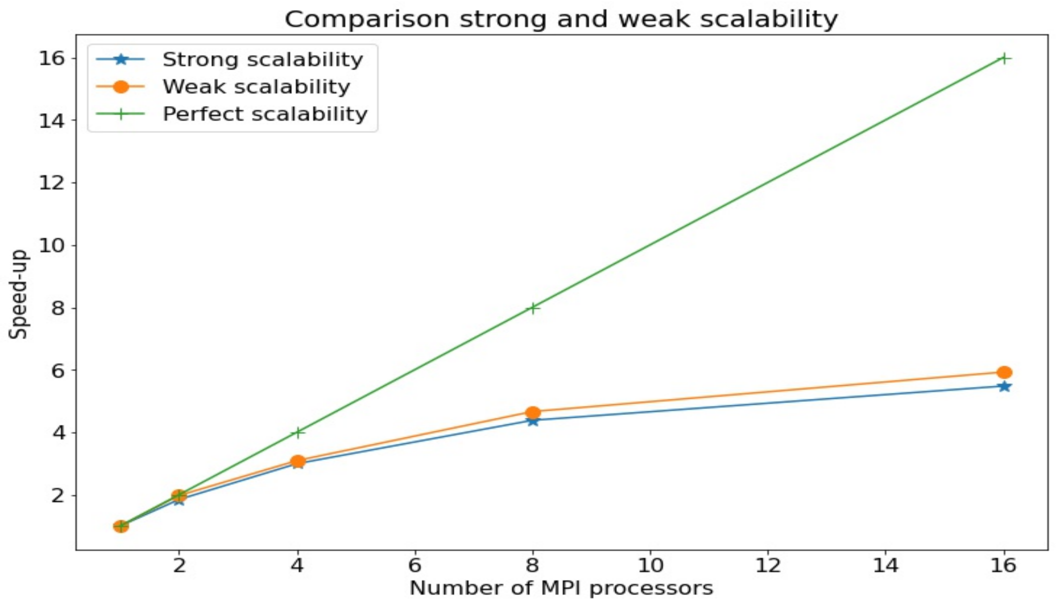

Figure 1.5: Comparison between scalabities and the perfect case scenario

1.5 Discussion

The decision to partially sort each time a local part of the total array is mainly because of the difference in performance respecting accessing memory and floating point operations, in the case of storing previous results in additional arrays will create a enormous overhead for huge arrays affecting the scalability of the implementation.
Along the runs it seems that the best performance is obtained when the number of threads is 2 and the rest are MPI processors, even though maybe increasing the task size threshold to reduce granularity could improve this.
As seen in weak scalability, is clearly visible that the percentage of parallel work increases with the number of points due to new levels of depth that could be exploited by adding more processors.

1.5.1 Points to improve

The selection of quick select or partial sorting it was a great asset but the performance could improved by just finding the way to parallelize this algorithm as its complexity is linear in the average case, which is one point to improve in this implementation.
Another approach tried on this implementation was to build the KDTree right away with only MPI where the problem its to avoid the overlapping of nodes creation which is a redundant work in this implementation and wasting MPI process time during the first nodes, this approach shows better performance for thousand of points and few threads because avoids the overhead of MPI communication, but as soon as the number of processors compensate this, the original Hybrid approach starts to outperform.

Another point to improve this implementation is trying to get rid of the work redundancy done by the OMP threads at the first nodes because each thread will share at the least the first node.
This implementation works to only perform for two OMP threads per MPI processor so a change in the creation of the master tree could revert this because that is where lays the heavy work of the whole implementation, so taking in consideration a way to create this master tree taking in consideration more than two threads and avoiding the work redundancy could improve a lot this code.

Bibliography

[1] Chopp, D. L. (2019). *Introduction to High Performance Scientific Computing (First ed.)*. SIAM - Society for Industrial and Applied Mathematics.

[2] Amdahl, G. M. (2007). *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. 1967, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California. IEEE Solid-State Circuits Newsletter, 12(3), 19–20. https://doi.org/10.1109/n-ssc.2007.4785615