Foundations of HPC - Assignment 2

Alessandro Masini

March 21, 2022

1 Introduction

A k-d tree is a space-partitioning data structure that allows to efficiently organize points in a k-dimensional space. In practice this can be implemented as a hierarchical structure where each element (or *node*) is connected to two children nodes (or *branches*), and an index representing the direction along which the children are split. such a structure can be built in an iterative fashion: one starts from from the root node, chooses a direction in the k-dimensional space, then splits the tree in two subtrees along said direction and defines the children as the roots of the left/right subtrees.

In this report we will provide a description of how a kdtree algorithm can be implemented using the C programming language, and two possible strategies that can be used to make it parallel. Finally we will analyze the performance of the code to find out how these strategies perform with respect to the serial version. All source code and data for this report can be found at this link.

2 Description of the algorithm

Assuming we already have the points of the tree, and that the size will remain fixed (i.e. there will be no further insertions or deletions), the steps that we need to build the k-d tree are mainly three: first we need to determine the split point from all the available points, second we have to split the tree in its left and right subtrees based on the split point chosen, and third we have to repeat the process for each subtree, in order to take their split points as children for the main tree point.

The most crucial part in this process is probably how we choose the split point. This influences the final result, and may give rise to balanced or unbalanced trees. In our case we use the median point as split, since this guarantees a balanced tree in the end (which would be better if we had to perform some searches on the tree after building it). The downside of this choice is that obtaining the median clearly requires some work. In our case we decided to use a quickselect algorithm, which is of order $O(n)$ in the best case, $O(n^2)$ in the worst. A negative side of this algorithm is that it is impossible to parallelize, since each step depends on the previous one. Nonetheless, it is generally really fast, thus this is not a bottleneck when parallelizing.

Another important decision, tightly bound to the choice of the split point, is how to choose along which of the possible k-dimensions we should sort and split the tree at each step. Our choice here is to round-robin through all the dimensions, starting from the first. This is quite simple, and does not require extra work when building the tree. We would obtain a more balanced tree if at each step we chose the direction of maximum extend of the data, but this in turn would require extra $O(n)$ work at each step.

All in all, the final shape of our algorithm turns out to be something like this:

1 - find median along chosen direction

2 - assign points to left/right subtrees

3 - update the dimension (round-robin)

4 - find median of left/right subtree along new dimension and set as left/right child

5 - if a subtree contains only one node, then stop

2.1 Implementation

As for the code implementation of the above algorithm, it is mainly based a custom type representing a tree node, which is defined as follows:

```
typedef struct kdnode kdnode_t;
struct kdnode {
    float split[NDIM];
    int axis, left, right;
};
```

The program reads datapoints from a provided file and stores them in a dynamically allocated array. Nonetheless, the number of dimensions (NDIM) and number of points (NPTS) of the tree must be known at compile time, thus they have to be either hard coded in the source file, or specified with appropriate definitions during compilation. The nodes' split points are initialized with the provided points, while axis, left and right are initialized to -1.

After reading the data, the tree is recursively generated using the following function:

```
int growTree(kdnode_t *tree, int start, int end, int axis)
{
    // if end == start then stop
    if (!(end-start)) return -1;

    // else do the recursive procedure
    int md;
    if ((md = findMedian(tree, start, end, axis)) >= 0 )
    {
        (tree+md)->axis = axis;
        axis = (axis+1) % NDIM;
        (tree+md)->left = growTreeSerial(tree, start, md, axis);
        (tree+md)->right = growTreeSerial(tree, md+1, end, axis);
    }
    return md;
}
```

As we can see this function simply computes the median, updates axis associated to the median node, then updates the value of the axis, and recursively calls itself on the left subtree (points up to the median) and right subtree (points after the median). Furthermore, if one passes zero points to the function (i.e. start==end), the function returns -1, meaning that the tree branch ends there. Note that the nodes do not contain pointers to locations in memory, but simple indexes, since we decided to store all nodes into an array. This is also more friendly when we have to parallelize this algorithm to distributed memory situations (MPI).

The **findMedian**(...) function implements the quickselect algorithm in order to find the median of an array of nodes. We avoid reporting the entire structure of the function, suffice to say that it repeatedly swaps nodes until all the nodes up to the median one are ordered, then it returns the index of the median. This takes care of the serial implementation. For the parallel implementations in MPI and OpenMP, the basic idea remains the same as the serial algorithm, the difference is that the **growTree**(...) function has to be modified to take advantage of the multiple cores/threads. In the following we report the structure of the function for both cases.

2.1.1 OpenMP parallelization

In the openMP the parallelization is performed using tasks. Instead of computing first the left subtree, then the right subtree, we spawn a task for each subtree, and the first available thread takes care of it. This

reduces the idle time of the threads and keeps them busy. The overall structure of the **growTree**(...) thus becomes

```
int growTree(kdnode_t *tree, int start, int end, int axis)
{
    // if end==start then stop
    if (!(end-start)) return -1;

    // else do the recursive procedure
    int n;
    if ((n = find_median(tree, start, end, axis)) >= 0 )
    {
        (tree+n)->axis = axis;
        axis = (axis+1) % NDIM;

        #pragma omp task
        {
            (tree+n)->left = growTree(tree, start, n, axis);
        }

        #pragma omp task
        {
            (tree+n)->right = growTree(tree, n+1, end, axis);
        }
    }
    return n;
}
```

The function has to be called inside an open parallel region, but since this would lead to race conditions due to multiple threads operating on the same memory portion, we have to wrap the function into a **single** clause. This ensures that a single thread accesses the same memory area at a time, and that each task is spawned only once.

```
int main(...)
{
    ...
    int root;
    ...
    // build tree
    #pragma omp parallel
    {
        #pragma omp single
        {
            root = growTree(tree, 0, NPTS, 0);
        }
    }
    ...
    return 0;
}
```

2.1.2 MPI parallelization

The MPI is a bit more complex. Since we do not have shared memory, we have to send data around. For this purpose we define a custom MPI datatype, which mimics the node structure

```
MPI_Datatype MPI_kdnode_t;
...
int main(...)
{
    ...
    // define custom MPI datatype for passing kdnode
    int blocklen[2] = {NDIM, 3};
    MPI_Aint disp[2] = {0, NDIM*sizeof(float_t)};
    MPI_Datatype oldtypes[2] = {MPI_FLOAT_T, MPI_INT};
    MPI_Type_create_struct(
        2, blocklen, disp, oldtypes, &MPI_kdnode_t);
    MPI_Type_commit(&MPI_kdnode_t);
    ...
}
```

All the data are initially allocated on the process with rank 0, and at each step we compute the median on rank 0, and send all the points after the median to the process with rank 1. After this, we split the MPI communicator in two (even ranks, odd ranks) and assign the left subtree to one subcommunicator, and the right subtree to the other, recursively calling the function. Finally, when we reach a communicator size of 1, the process becomes serial. When each process has grown the tree from the data it received, it sends back the data to up to process 0. In the end all data comes back to root process in an ordered fashion, and the tree is completed. The crucial part of the function is reported below (the whole function can be found in MPI version of the code on the GitHub repository):

```
int growTree(kdnode_t *tree, int start, int end, int offset, int axis, MPI_Comm comm)
{
    int nright = -1, nleft = -1;
    ...
    // split communicator
    MPI_Comm new_comm;
    splitComms(comm, &new_comm);
    if (comm_rank % 2 == 0)
    {
        nleft = growTree(tree, 0, md, offset, axis, new_comm);
    }
    else
    {
        nright = growTree(tree, 0, right_count,  offset+md+1, axis, new_comm);
    }
    ...
}
```

Here we can see how we split the communicator and then perform half of the tree construction in one communicator, and the other half in the other communicator.

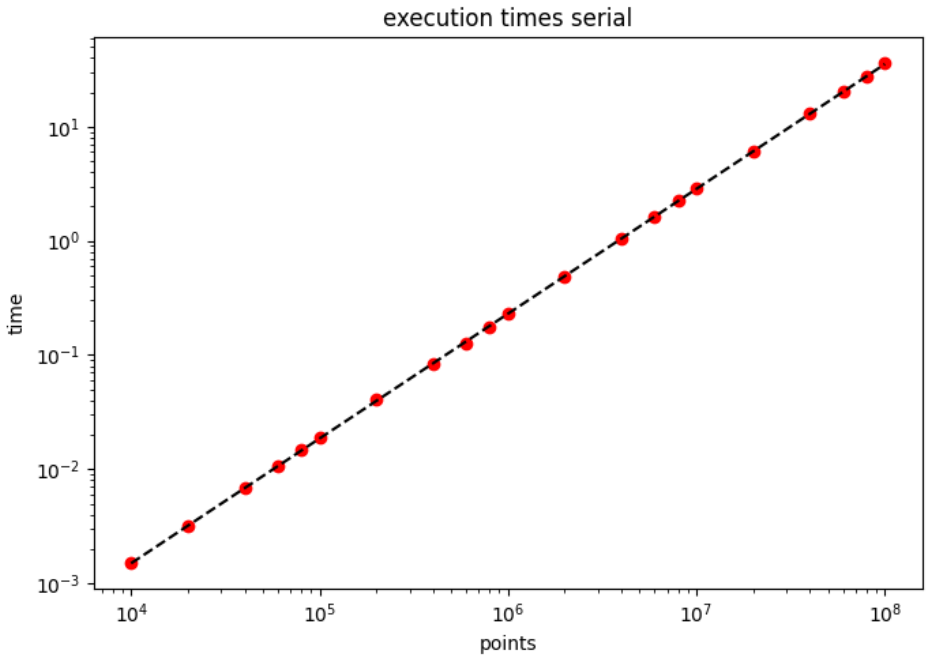

Figure 1: Execution times for the serial algorithm (log-log plot).

3 Performance model for serial algorithm

In this section we will provide a brief analysis for the performance of the serial algorithm, in order to understand how it behaves, and which kind of performances we can expect from it. In order to build our model, we ran our serial code on different sizes of point sets, ranging from 10^5 to 10^8. For each size we performed 5 runs, then averaged the measured times. The results are shown in figure 1. As we can see, using logarithmic axes the execution times lie on a straight line. From this we can conclude that the following relation holds:

$$\log T(N) \simeq c \log N + d\,,$$

and thus

$$T(N) \simeq e^d N^c\,,$$

i.e. the time scales as some power of the size of the data set. To find out the exponent c we can perform a polynomial fit on the logarithms of the times and the sizes. The resulting values for the data plotted above are $c = 1.0929$, $e^d = 6.3774e-08$, thus we can conclude that the execution time grows (almost) linearly with size, with a very small proportionality factor (that is, up to 10^8 points).

4 Scalability

After a brief check on the performance model for the serial algorithm, we now have to discuss how well the algorithm scales when we parallelize it. We will discuss both OpenMP and MPI versions at the same time, in order to compare the two different implementations.

4.1 Strong scalability

In order to check the strong scalabilty of our algorithms, we ran the codes on a point set of 10^8 points, using a growing number of processes (up to 32 tasks for MPI, up to 16 threads for OpenMP). For each number of tasks we performed 5 runs, then averaged the obtained execution times. The final results are shown in the tables 1. We also computed the speedup, the efficiency and the serial and parallel fractions (estimated). The speedup is simply computed as $s(n) = t(1)/t(n)$, while the efficiency is $s(n)/n$. The parallel fraction

n	time	$s(n)$	ε_s	S_f	P_f
1	36.45150	1.00	1.00	1.00	0.00
2	22.16010	1.64	0.82	0.22	0.78
4	14.39090	2.53	0.63	0.19	0.81
8	10.51180	3.47	0.43	0.19	0.81
16	8.80113	4.14	0.26	0.19	0.81
32	7.83328	4.65	0.15	0.19	0.81

n	time	$s(n)$	ε_s	S_f	P_f
1	36.45150	1.00	1.00	1.00	0.00
2	22.12140	1.65	0.82	0.21	0.79
4	13.68940	2.66	0.67	0.17	0.83
8	9.88588	3.69	0.46	0.17	0.83
16	9.48660	3.84	0.24	0.21	0.79

Table 1: MPI (left) and OpenMP (right) strong scaling data.

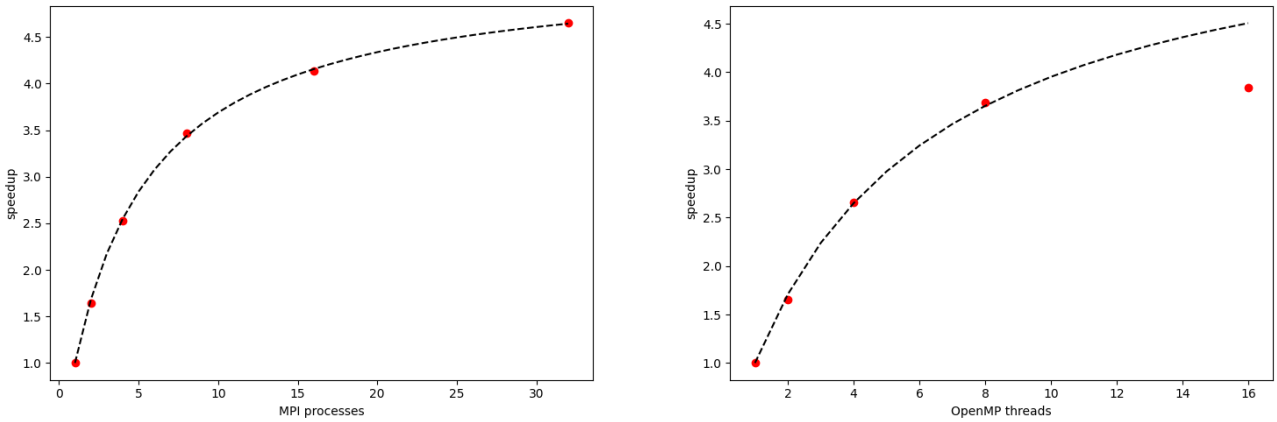

Figure 2: MPI (left) and OpenMP (right) expected and mesured speedups.

P_f, on the other hand, was obtained from the speedup by inverting Amdahl's law:

$$s(n) = \frac{1}{(P_f/n) + (1 - P_f)} \quad \rightarrow \quad P_f = \frac{[s(n) - 1]\, n}{s(n)\,(n-1)}$$

Finally, the serial fraction is computed as $S_f = 1 - P_f$.

As we can see from the tables above, the two different parallelization strategies are almost equal if we only consider the numbers. Up to 16 cores the strong scaling efficiency is almost the same for both OpenMP and MPI, thus we cannot decide which algorithm is better. If we only consider the speedup, on the other hand, we can see that the MPI version has a better scaling when passing from 8 to 16 tasks (where the execution time for the OpenMP version is almost unchanging).

As for the parallel fractions estimated using Amdahl's law, we can see that they are similar for both algorithms. In figure 2 we can see a comparison of the expected and measured speedups, if we take as true value for the parallel fraction, the highest of the values obtained from the speedups (i.e. the max in the relative column of the table). We can clearly see that the OpenMP version scales well up to 8 threads, and after that value the effective speedup is much lower than the expected value.

4.2 Weak scalability

In order to analyze the weak scalability of the code, we performed a few runs increasing the number of parallel tasks, but keeping the amount of points per task fixed to 10^7. For each size/number of tasks we performed 5 runs, then averaged the collected execution times. We also collected execution times from the serial code for the same tree sizes, in order to compute the speedup, from which we obtained the parallel and serial fraction using Gustafson's law:

$$s(n) = \frac{t_s}{t_p} = S_f + n\, P_f \quad \rightarrow \quad P_f = \frac{s(n) - 1}{n - 1}\,, \quad S_f = 1 - P_f\,.$$

Ideally, as per Gustafson's law, one would expect a linear (or close to linear) speedup in a weak scalability situation. However, as we can see from tables 2 and 3, this is totally not the case. The reason for this is

n	N	t_s	t_p	$s(n)$	ϵ_w	S_f	P_f
1	10000000	2.88946	2.88946	1.00	1.00	1.00	0.00
2	20000000	6.00992	3.54410	1.70	0.82	0.30	0.70
4	40000000	12.94810	4.75879	2.72	0.61	0.43	0.57
8	80000000	27.53880	6.91959	3.98	0.42	0.57	0.43
16	160000000	56.72220	10.84290	5.23	0.27	0.72	0.28
32	320000000	121.62100	20.88690	5.82	0.14	0.84	0.16

Table 2: MPI weak scaling data.

n	N	t_s	t_p	$s(n)$	ϵ_w	S_f	P_f
1	10000000	2.88946	2.88946	1.00	1.00	1.00	0.00
2	20000000	6.00992	3.63097	1.66	0.80	0.34	0.66
4	40000000	12.94810	4.48512	2.89	0.64	0.37	0.63
8	80000000	27.53880	6.32767	4.35	0.46	0.52	0.48
16	160000000	56.72220	10.62470	5.34	0.27	0.71	0.29

Table 3: OpenMP weak scaling data.

that Gustafson's law is based on the assumption that the serial part of the problem does not grow when we increase the problem size. This is untrue in our case, we can easily prove it if we realize that the first iteration of our algorithm is performed by one processor only, and the amount of work needed to find the median of the whole tree grows with the size of the tree. This explains the growing (shrinking) serial (parallel) fraction which we see in the tables.

Probably the most interesting metric in this case is the weak efficiency. As we can see it is decreasing, as we expected, since increasing the number of parallel task we gain relatively less and less performance. Figure 3 shows a comparison of the MPI and OpenMP parallelizations based on their parallel efficiency. As we can see OpenMP has a slightly better efficiency for intermediate values of the number of tasks, but for $n = 16$ the value is exactly the same for the two implementations. In general the OpenMP implementation seems to have a better weak scaling than the MPI, but it is nothing significant.

5 Conclusions

To conclude our discussion, from our data we can see that the OpenMP implementation has a better scaling (both weak and strong) than the MPI version when using up to 8 tasks. After this value, the OpenMP implementation seems to stop scaling. This is almost certainly due to the code not being very well optimized. It may be useful to restructure a bit the code, in order to allow a better distribution of the workload between the parallel tasks, and manually decide which process acts on a specific part of the node vector, instead of relying on OpenMP tasks. This may result in less frequent thread switching, and a global reduction of the time between two iterations of the algorithm.

The MPI implementation can be optimized as well. The crucial thing to notice is that the paradigm we adopted in parallelizing the code is more suited for a shared memory situation. Even though we obtain comparable results with the MPI version, it would probably be better to adopt a distributed memeory approach to the kdtree, for example distributing the data in the beginning, and sending data between processors in order to reorder them at each iteration. This may even augment the number of communications needed, but it would also allow a linear weak scaling, since the serial part of the code remains fixed when increasing the workload. Furthermore, this would be the perfect starting point for an hybrid MPI/OpenMP implementation, where each MPI task is delegated to more than one thread, which acts in our current OpenMP implementation.

This would be probably the best implementation, allowing a faster tree build-up, much better scaling with the tree size, and removing possible memory bottlenecks due to the memory of a single node not being

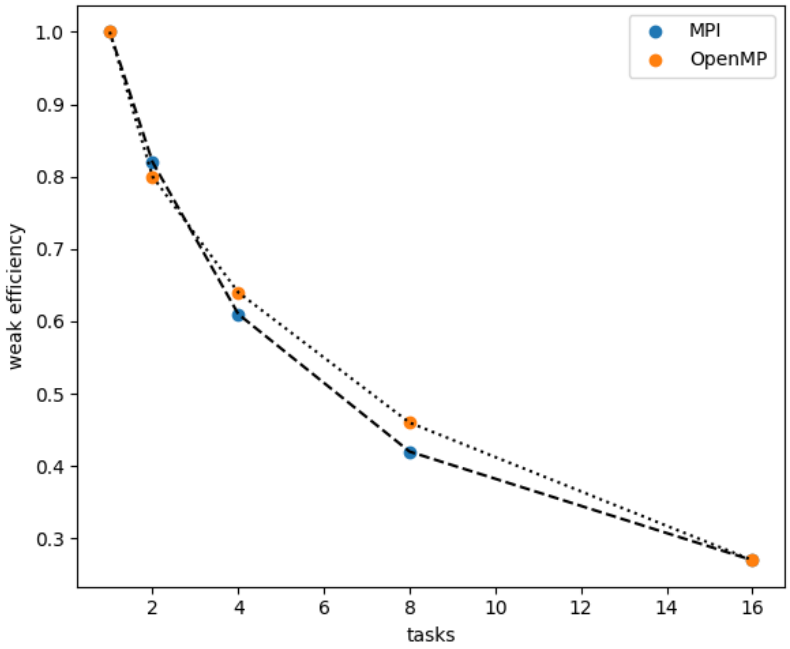

Figure 3: Parallel efficiency MPI vs OpenMP

sufficiently large to contain all the data we need.