github.com/SilvioBaratto/KD-tree

Parallel k-d tree OpenMP & OpenMPI

Silvio Baratto

17 maggio 2022

Indice

1 Introduction

In computer science, a k-d dimensional tree is a space-partitioning data structure for organizing points in a k-dimensional space. k-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches) and creating point clouds. k-d trees are a special case of binary space partitioning trees. In this brief report is presented a parallel implementations of k-d tree which uses two standard frameworks for parallel programming, namely OpenMP (shared memory) and MPI (distributed memory).

2 Algorithm

The construction of the tree is done by:

- Finding median / pivot by medians of the input file. Each section of constant lenght is sorted bt $std :: nth\_element$ from the C++ standard library.
- Recursively proceed on left and right portions on the left and right of the found median.
- Terminate when length of a portion is 0.
- Return root node.

The time complexity of the divide and conquer algorithm is $O(nlog(n))$ since the partition process always picks the middle element as pivot. The median of the medias is found in linear time, $O(n)$.
Time complexity for partitioning n datapoints:

$$T(n) = 2T(n/2) + \theta(n)$$

2.1 Implementation

The KD-Tree implementation was made with C++17 and written using C++ templates so is a generic programming algorithm working with both float, integers and double numbers. The purpose of this code was to be as simple as possible to use from the point of view of a user. The goal was to generate a k-d tree from a file choosen from a local directory of the user, containing points in two dimension. To do this, the constructors of all the three classes (mpi, omp, serial) creates a non ordered vector of knodes using as points the ones in the dataset, in this way it has been possible also to use the standard library functions provided from C++17.
The implemented algorithm doesn't use a sorting algorithm, but it uses the function provided by the standard library "$std :: nth\_element()$". This function is a partial sorting algorithm that rearranges elements in [first, last) such that:

- the element pointed at by nth (the median) is changed to whatever element would occur in that position if [first, last) were sorted.
- All of the elements before this new nth element (the median) are less than or equal to the elements after the new nth element. With this function, it's possible to omit the implementation of a sorting algorithm because it has the best "worst expected running time", that is $O(N)$.

Algorithm 1 makeTree

```
1: function MAKETREE(begin, end, axis)
2:     if end ≤ begin then                                   ▷ base case
3:         return nullptr
4:     end if
5:     median ← begin + (end - begin) / 2
6:     NTH_ELEMENT(begin, med, axis, cmp(axis))
7:     myaxis ← round robin approach between 0 and 1
8:     knode[med].left ← MAKETREE(begin, med, myaxis)
9:     knode[med].right ← MAKETREE(med + 1, end, myaxis)
10:    return &knode[med]
11: end function
```

2.2 OpenMP

OpenMP is one of the application programming interfaces that facilitates the employment of a shared memory paradigm for parallelization within a node. Below the algorithm used in the implementation

Algorithm 2 makeTree

```
1: function MAKETREE(begin, end, axis)
2:     if end ≤ begin then                                   ▷ base case
3:         return nullptr
4:     end if
5:     median ← begin + (end - begin) / 2
6:     NTH_ELEMENT(begin, med, axis, cmp(axis))
7:     myaxis ← round robin approach between 0 and 1
8:     #pragma omp task
9:     knode[med].left ← MAKETREE(begin, med, myaxis)
10:    #pragma omp task
11:    knode[med].right ← MAKETREE(med + 1, end, myaxis)
12:    return &knode[med]
13: end function

14: function MAKETREEPARALLEL(begin, end, axis)
15:    #pragma omp parallel
16:    #pragma omp single
17:    root * ← MAKETREE(begin, end, axis)
18:    return root
19: end function
```

2.3 OpenMPI

Algorithm 3 makeTreeParallel

```
1: function MAKETREEPARALLEL(begin, end, axis, nprocs, depth, comm, next)
2:     if end ≤ begin then                                              ▷ base case
3:         return nullptr
4:     end if
5:     median ← begin + (end - begin) / 2
6:     NTH_ELEMENT(begin, med, axis, cmp(axis))
7:     myaxis ← round robin approach between 0 and 1
8:     if rank != 0 then
9:         if nprocs/2! = 2^depth then
10:            depth ← depth + 1
11:            knode[med].left ← MAKETREEPARALLEL(begin, med, myaxis, myaxis,
    nprocs, depth, comm, next)
12:            next = next + 2
13:            knode[med].left ← MAKETREE(med + 1, end, myaxis, nprocs, depth, comm,
    next)
14:        else
15:            if rank == next then
16:                knode[med].left ← MAKETREE(begin, med, index)
17:                serializeLeft ← SERIALIZENODE(knode[med].left)
18:                MPI_Send(serializeLeft, length, MPI_FLOAT, 0, left_tag, comm)
19:            end if
20:            next ← nprocs - 1 ? next + 1 : 1
21:            if rank == next then
22:                knode[med].right ← MAKETREE(med + 1, end, index)
23:                serializeRight ← SERIALIZENODE(knode[med].right)
24:                MPI_Send(serializeRight, length, MPI_FLOAT, 0, right_tag, comm)
25:            end if
26:        end if
27:    else if rank == 0 then
28:        if nprocs/2! = 2^depth then
29:            depth ← depth + 1
30:            knode[med].left ← MAKETREEPARALLEL(begin, med, index, nprocs, depth,
    comm, rank)
31:            next ← next + 2
32:            knode[med].right ← MAKETREEPARALLEL(med + 1, end, index, nprocs,
    depth, comm, rank)
33:        else
34:            MPI_Probe(next, left_tag, comm, &status)
35:            MPI_Get_Count(&status, MPI_FLOAT, &count)
36:            MPI_Recv(buffer_left, count MPI_FLOAT, next, left_tag, comm, &status)
37:            knode[med].left ← DESERIALIZE(buffer_left)
38:            next ← nprocs - 1 ? next + 1 : 1
39:            MPI_Probe(next, right_tag, comm, &status)
40:            MPI_Get_Count(&status, MPI_FLOAT, &count)
41:            MPI_Recv(buffer_right, count MPI_FLOAT, next, left_tag, comm, &status)
42:            knode[med].left ← DESERIALIZE(buffer_right)
43:        end if
44:    end if
45:    return &knode[med]
46: end function
```

The code to implement the OpenMPI is quite different from the OpenMP. The strategy has been to divide for each process a sub-tree and combined them in the main process. The process 0 merge all the sub-trees and construct the final one. When the the half of the number of process is equal to the power of 2 to depth each process works alone creating the sub-tree in a serial way and sending it to the master process with rank 0. To send and receive the messagges has been used a serialize and deserialize function that increase a lot the performance time and has been the main problem in this implementation.

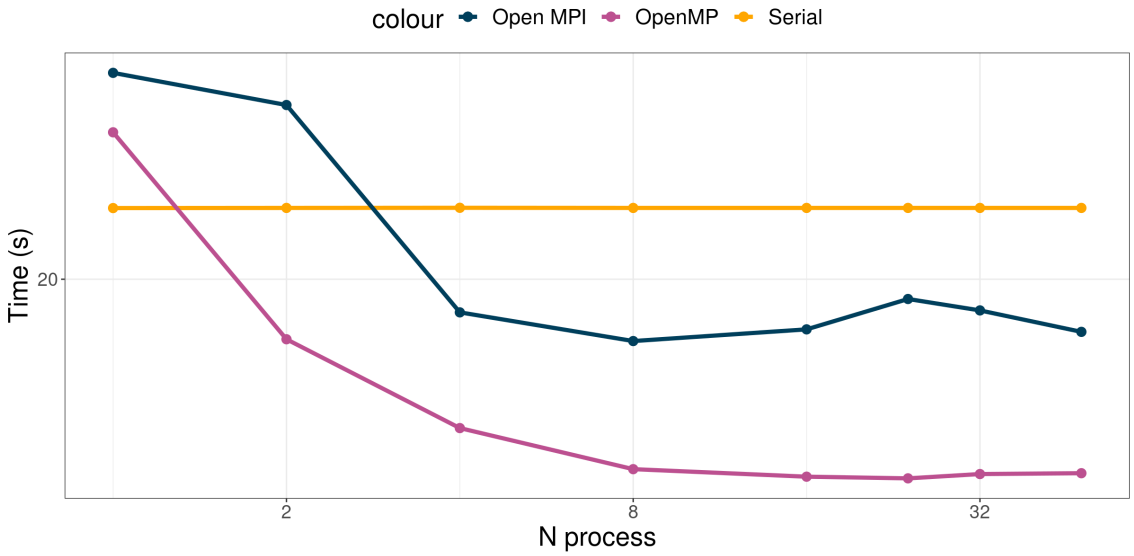
3 Strong and weak scalability

3.1 Strong scalability

The strong scalability explain the performance of the parallel algorithm with the number of processors that increase while the problem size remains constant. To study the strong scalability it has been used:

- $N = 100000000$
- $p \in \{1, 2, 4, 8, 16, 24, 32, 48\}$

From the graph above it's possible to see an huge difference between OpenMPI and OpenMP. The main problem of the OpenMPI implementation is the serialization and deserialization part of the code.

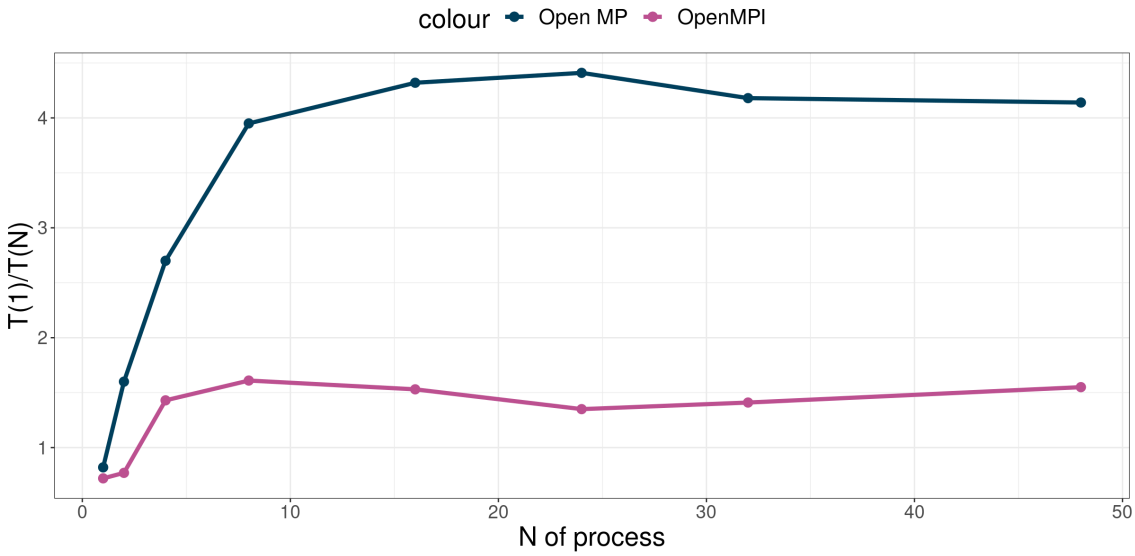


(a) Comparison between different implementation

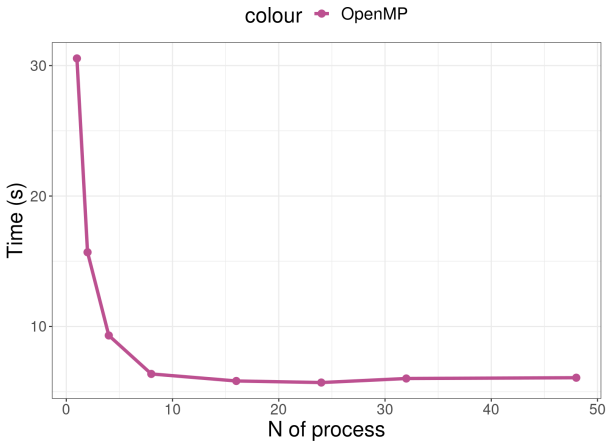
The speedup is the ratio of the one-CPU execution time to the N-CPU parallel execution time: $S(P) = T(1)/T(P)$. When $S(P) = P$ it is considered a perfect speed-up. This in "real-life" is not an ultimate goal, since the code usually has parts that need to execute in serial. This happens because some parts of a program can't be executed in parallel. This phenomenon is describes by the Amdahl's Law, that is:

$$S(P) = \frac{1}{(1-P)/(P/N)} \tag{1}$$

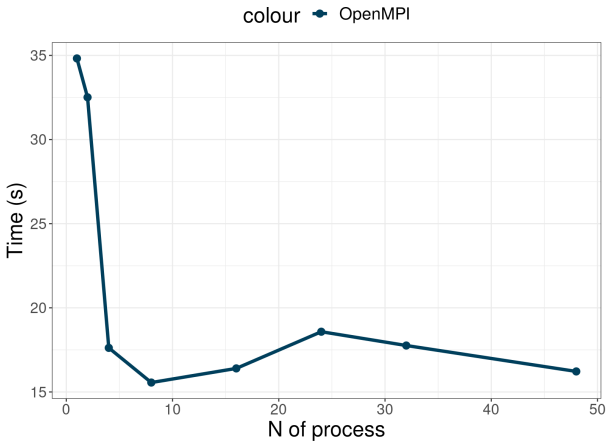
From the graph above OpenMP has best performance but Open MPI algorithm has greater margin for growth without serialization and deserialization.



(b) Speedup OpenMP and OpenMPI



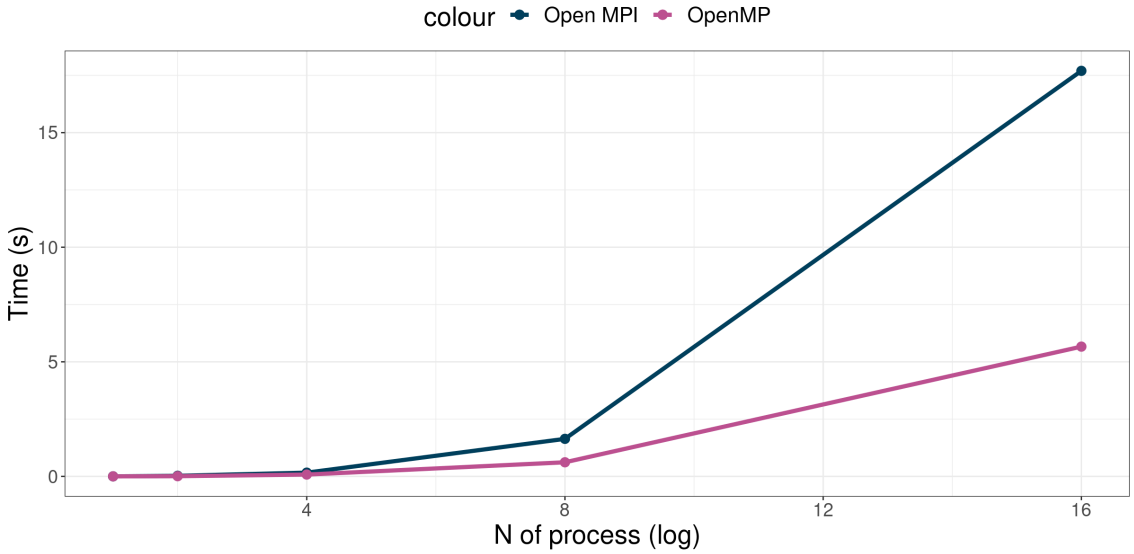
(c) Strong scalability OpenMP



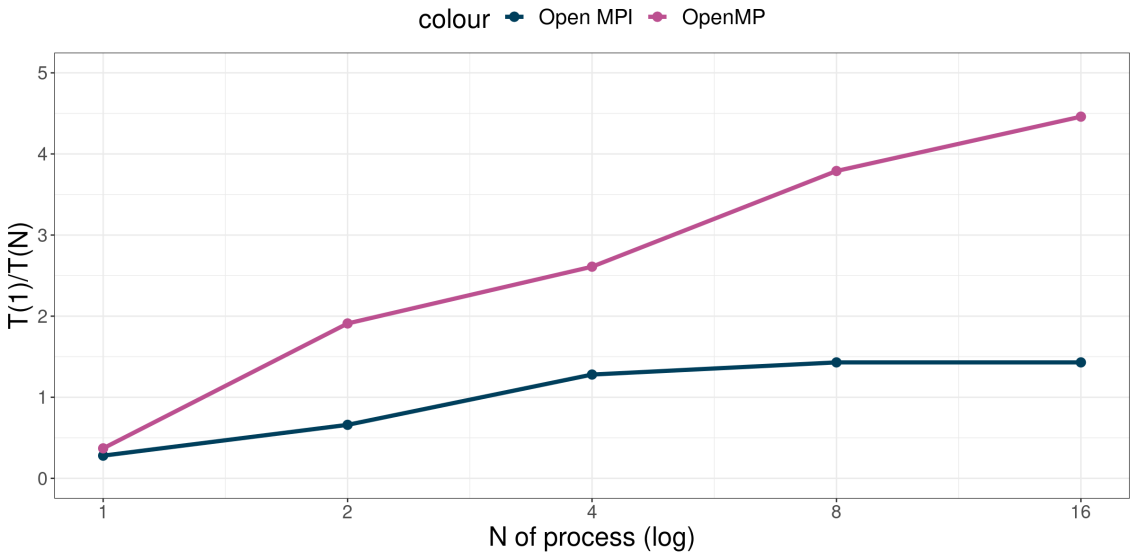
(d) Strong scalability OpenMPI

3.2 Weak scalability

In case of weak scalability, both the number of processes and the problem size are increased. The weak efficiency can be calculated by changing the problem size with each change of resources. The problem size was increased by a factor of 10 (starting from 10^5), with each scale up in resources. Also in this case is possible to see how clearly the OpenMP implementation performe better with higher size.



(e) Weak scalability OpenMP & OpenMPI



(f) Speedup OpenMP and OpenMPI

4 Conclusion

From this work is possible to see how OpenMP performes well the work in the construction of the k-d tree. In the other hand clearly OpenMPI must be improved and a possible solution to this is by removing the entire serialization and deserialization part sending instead the entire sub-tree to the master process. From the user point of view the final OpenMP solution has a better proportion between readability / performance. To improve the overall efficiency of the code a future implementation can be growing the k-d tree directly from the file taken in input without using *std* :: *vector* and to add some other functions like adding and deleting knodes.