Università degli studi di trieste

Assignment 2 KD-Tree

Silvio Baratto

18 marzo 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 nth element.
- Recursively proceed on left and right portions on the left and right of the found median. Each median is a node.
- 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 provide a k-d tree given only a filename of points choose from a local directory of the user. Because of this in 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 the implementation the tree is created from the vector, in this way it has been possible to use the standard library functions easily.
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:     MEDIANOFMEDIANS(begin, med, 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:     MEDIANOFMEDIANS(begin, med, 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:    root
16:    #pragma omp parallel
17:    #pragma omp single
18:    root ← MAKETREE(begin, end, axis)
19:    return root
20: end function
```

2.3 OpenMPI

The strategy used to implement the OpenMPI part was to divide for each process, a sub-tree that will be merged in only one process.
The process from the 1 to N construct a sub-tree to send at the main process. The work

of the process 0 is primarly to merge all the sub-trees and to construct the final kd-tree. Untill a good level of the final tree is reached, the group of processes build the tree in a serial way. A good level is reached when the half of the number of process involved is equal to the power of 2 to level. When you get to this level, each process works alone creating in a serial way the sub-tree and send it to the master process, with the rank 0. There is a shared variable that assign the work of each process, that is has to work in the right or in the left. If the number of process is smaller that the number of the sub-tree that has to built, this variable automatically assign which process has to work two times. Meanwhile, there is the process with rank 0, that does the first part like others processes and then waits to receive the messagges with all of sub-trees.
The main problem of this implemented code with OpenMPI is that each sub-tree has to be serialized in a string before and deserialize the message received after.

Algorithm 3 makeTreeParallel

```
1: function MAKETREE(begin, end, axis)
2:     if depth == log2(nprocs) then                                  ▷ base case
3:         return nullptr
4:     end if
5:     median ← begin + (end − begin)/2
6:     MEDIANOFMEDIANS(begin, med, axis)
7:     myaxis ← round robin approach between 0 and 1
8:     if rank! = 0 then
9:         if depth == log2(nprocs) 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_CHAR, 0, left_tag, comm)
19:                serializeRight < −SERIALIZENODE(knode[med].right)
20:                MPI_Send(serializeRight, length, MPI_CHAR, 0, right_tag, comm)
21:            end if
22:        end if
23:    end if
24:    if ( thenrank == 0)
25:        if depth == log2(nprocs) then
26:            depth ← depth + 1
27:            knode[med].left ← MAKETREEPARALLEL(begin, med, index, nprocs, depth,
    comm, rank)
28:            next ← next + 2
29:            knode[med].right ← MAKETREEPARALLEL(med + 1, end, index, nprocs,
    depth, comm, rank)
30:        else
31:            MPI_Probe(next, left_tag, comm, &status)
32:            MPI_Get_Count(&status, MPI_CHAR, &count)
33:            MPI_Recv(buffer_left, countMPI_CHAR, next, left_tag, comm, &status)

34:            knode[med].left < −DESERIALIZEPARALLEL(buffer_left)
35:            MPI_Probe(next, right_tag, comm, &status)
36:            MPI_Get_Count(&status, MPI_CHAR, &count)
37:            MPI_Recv(buffer_right, countMPI_CHAR, next, left_tag, comm, &status)

38:            knode[med].left < −DESERIALIZEPARALLEL(buffer_right)
39:        end if
40:    end if
41:    return &knode[med]
42: end function
```

3 Performance

4 Conclusion