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

Algorithm 3 makeTreeParallel

```
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:    if rank! = 0 then
11:        if nprocs/2! = pow(2, depth then
12:            depth = depth + 1
13:            knode[med].left ← MAKETREEPARALLEL(begin, med, myaxis, myaxis,
    nprocs, depth, comm, which)
14:            next = next + 2
15:            knode[med].left ← MAKETREE(med + 1, end, myaxis, nprocs, depth, comm,
    which)
16:        else
17:            if rank == next then
18:                #pragma omp parallel
19:                #pragma omp single
20:                knode[med].left ← MAKETREE(begin, med, index)
21:                knode_to_string < −SERIALIZENODE(knode[med].left)
22:            end if
23:        end if
24:    end if
25:    return &knode[med]
26: end function
```

3 Performance

4 Conclusion