CS 426/525 Fall 2021 Project 3
k-d Trees

Due 5/12/2021, 23:59

1 Introduction

In this project, we will focus on k-d Trees as a space-partitioning data structure. As usual, you will first implement a serial program and then you will implement the parallel version of it. The program will create a k-d tree and process range search queries on this k-d tree. The parallel version of it should be implemented using the OpenMP library by inserting *pragma* directives on the serial version. Similar to the previous projects, you are going to use C or C++ programming languages. You will compare parallel and serial versions with various inputs, then write a short report of your findings. Submissions will be on Moodle.

2 k-d Trees

k-d Trees are binary trees that partition the points in a k-dimensional space. They are used for range searches that involve multiple key values. For example, assume there is a database of students and each student has a name, age, id number, and grade in this database. Furthermore, the query to be executed is to get the students whose ages are between 20 and 25, id numbers are between 1000 and 2000, and grades are between 3.00 and 4.00. A k-d tree structure would be very useful for getting the students for the given range. k-d trees are also used for Nearest Neighbor Search (NNS) problems and for Point Clouds. For this assignment, we will focus only on creating a k-d tree and range search problem.

We first describe creating a k-d Tree. For a given set of points in k-dimensions, we create a binary tree such that all of the left children of a parent node are smaller than the parent in one of the k dimensions. Similarly, all of the right children of the parent node are greater than the parent for the same dimension. For the chosen dimension, the *term* of the parent's point is the median among terms of the left and the right points. An example k-d tree is shown in Figure 1 for the set of points:

$\{(83, 20, 32), (3, 50, 94), (23, 15, 56), (21, 41, 53), (72, 7, 25), (58, 91, 32), (10, 95, 63), (29, 99, 4)\}$

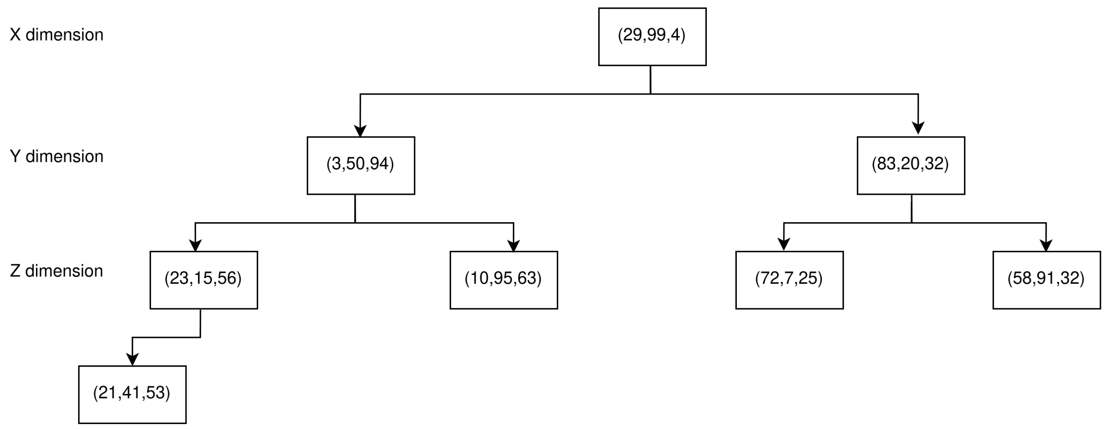

Figure 1: Sample k-d Tree

At each level of the tree, we cyclically change the dimension. At the very top level, the point $(29, 99, 4)$ is chosen as the root because 29 is the median value of the terms in the x dimension. All points whose term value for the x dimension is less than 29 should be at the left of the root on the binary tree and all points whose term value for the x dimension is greater than 29 should be at the right of the root. At the next level, we do the same but for the y dimension. For example, all the left children of the node $(3, 50, 94)$ has y-terms that lass than 50. Construction of a k-d tree is recursive. A recursive algorithm for the construction is given in Algorithm 1 along with an illustration in Figure 2.

Algorithm 1: k-d Tree Construction Algorithm

```
Input  : List of points $P$, $dimension$
Output: Root node $R$
1
2 $kdTreeConstruct(P, dimension, 0)$
3
4 function kdTreeConstruct(Set of points $P$, dimension, $k$):
5     Let $N$ be a binary tree node
6     Select median point $M \in P$ where key value is $k$
7     $N.data \leftarrow M$
8     if $|P| \leq 1$ then
9         return $N$
10    end
11    $Pleft \leftarrow \{ x \mid x \in P$ and $x[k] \leq M[k]$ and $x \neq M \}$
12    $Pright \leftarrow \{ x \mid x \in P$ and $x[k] \geq M[k]$ and $x \neq M \}$
13    $k \leftarrow (k+1) \mod dimension$
14    $N.left \leftarrow kdTreeConstruct(Pleft, dimension, k)$
15    $N.right \leftarrow kdTreeConstruct(Pright, dimension, k)$
16    return $N$
```

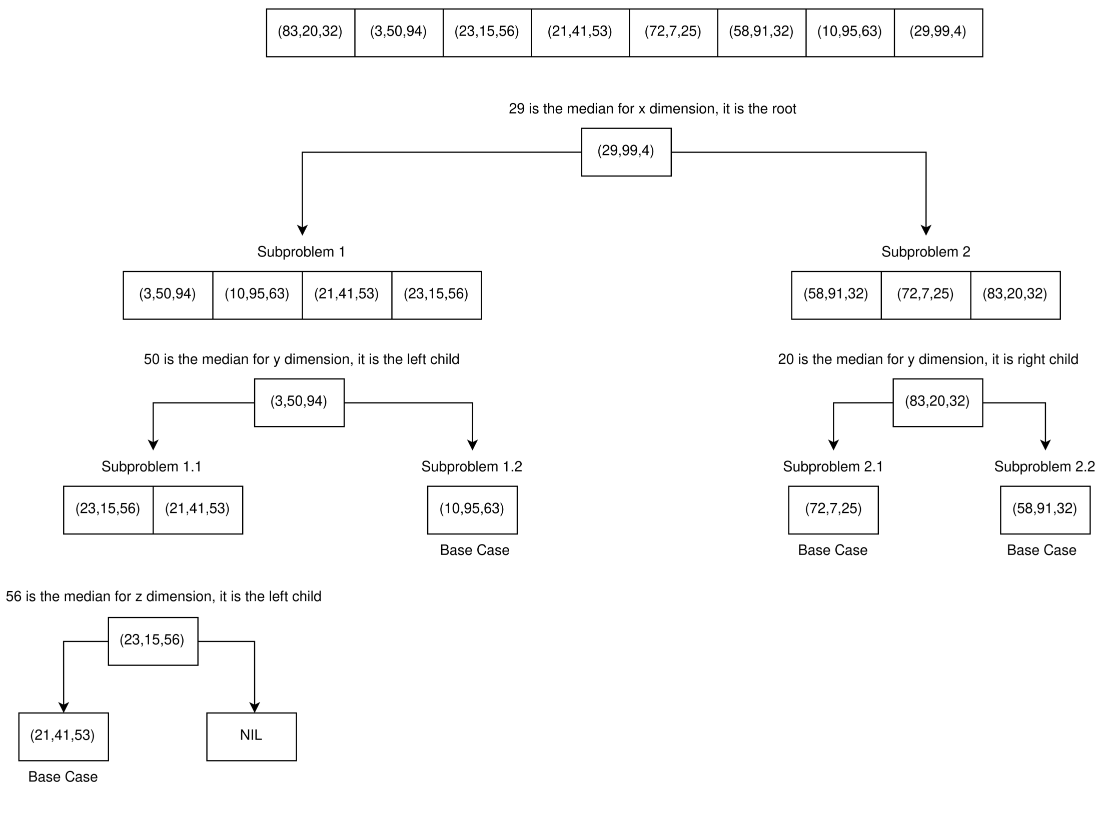

Figure 2: Construction of the Sample k-d Tree

A range search algorithm on k-d data structure is given in Algorithm 2. It is also a recursive algorithm similar to the construction. For details on k-d trees and geometric search problems, refer to Chapter 2 in [2].

Algorithm 2: Algorithm for Range Search on k-d Tree

```
Input  : root of $k$-d tree $N$,range $R$ $dimension$
Output: points $P$

$P \leftarrow \{\}$
$rangeSearch(N, dimension, 0, R, P)$

function $rangeSearch$(root of $k$-d tree $N$, dimension, $k$, range $R$, result points $P$):
    if $N \neq NIL$ then
        if $N.data[i] \geq range.left[i]$ and $N.data[i] \leq range.right[i], \forall i\ 0 \leq i \leq dimension$ then
            add $N.data$ to $P$
        end
        $k_{next} \leftarrow (k+1) \mod dimension$
        if $N.data[k] \geq R.left[k]$ then
            $rangeSearch(N.left, dimension, k_{next}, range, P)$
        end
        if $N.data[k] \leq R.right[k]$ then
            $rangeSearch(N.right, dimension, k_{next}, range, P)$
        end
    end
```

3 Serial Implementation

The first part of the project is to implement the construction of k-d tree and range search algorithm as described in Section 2. Your program should take an input file name for the list of points, another input file name for the list of ranges, and an output file name as arguments.

For a range query, you should find how many points exists in range in the k-d tree. For example, a range query of $[(5, 1, 17), (80, 70, 90)]$ for the sample k-d tree given in Figure 1 has points $\{(23, 15, 56), (21, 41, 53), (72, 7, 25)\}$. Then number of points for the range is 3. The number of points for each range in queries should be written to the output file in order. The input file for the list of points should be an ASCII text file, whose first line gives the number of points m, the second line gives the number of dimensions d, each of the next m lines gives the d terms for a point. In each line, d terms are separated by a space. The input file for range queries should be an ASCII text file, whose first line gives the number of queries q, the second line gives the number of dimensions d, each of the next q lines gives the $2 \times d$ terms for two points in the range where each term is separated by a space. The first d terms are for the left point of the range and the second d terms are for the right point.

The output file will be an ASCII text file as well, consisting of q (number of queries) lines, line i gives the number of points in the range i in the list of queries. Sample inputs and sample outputs are given within this document for the examples above. The name of the program should be `kdtree_serial` Following illustrates how your program should take the arguments:

```
# ./kdtree_serial ANY_POINTS_INPUT_FILENAME.txt ANY_RANGE_INPUT_FILENAME.txt ANY_OUTPUT_FILENAME.txt
```

For reading the input files and outputting the results, you may use the `util.h` and `util.c` files we provided. There are also some other utility functions that you may find useful in these utility files.

For median finding, an easier way would be sorting the points for the key values at a given dimension and getting the element in the middle index. This would be an O(nlogn) algorithm. But there are expected linear time algorithms that you may find in Chapter 9 of [1]. You are welcome to use them[1] for your implementations.

4 Parallel Implementation

You will use OpenMP library to parallelize your *kdtree* program by inserting *pragma* directives into your code.

An analysis of your program, such as loops, recursive calls would be useful for inserting the proper directives. You should use performance tools for analysis. We will use *perf* tool[2] for doing so. Namely, we will get the call graph of the program using this tool to analyze which parts of the application consumes the most CPU resources. This would be useful for focusing on which parts we should make parallel. An example usage of *perf* is given below for a code that takes a single input

[1] Also see https://en.wikipedia.org/wiki/Selection_algorithm
[2] https://en.wikipedia.org/wiki/Perf_(Linux)

argument in command line:

```
# g++ -fno-omit-frame-pointer main.cpp -o test [3]
# sudo perf record -g test input.txt
# perf report --stdio -g fractal,callee --no-children -i ./perf.data > analysis.txt
```

You should run the commands for *perf* similar to above for the *kdtree* program. If there are issues with *perf* tool in your system for some reasons, you can also try *gprof* tool. Note that, you may see some weird and deep-nested results for the function calls due to recursive functions. Try to focus on top-level function usage in the call-graph, as well as your own reasoning.

Try inputs with large number of points for k-d tree construction and small number of range queries to analyze the k-d tree construction performance. For query performance, set a relatively small number of points for the k-d tree but large number of range queries. You may also set both inputs to sufficiently large values to see the actual performance effects. After your analysis, copy your serial program into a new file and insert your *pragma* directives.

Arguments of the parallel program should be the same as the serial version. The input/output file formats are also the same. The name of the program should be `kdtree_parallel`. Following illustrates how to call your program with the respective arguments:

```
# ./kdtree_parallel ANY_POINTS_INPUT_FILENAME.txt ANY_RANGE_INPUT_FILENAME.txt ANY_OUTPUT_FILENAME.txt
```

5 Assumptions

- For the parallel version of the program, you may change your function signatures in the serial part and add new functions if you think it is necessary.

6 Hints

- As we are only interested in the number of points for a range query, you don't need to compute the resulting points.
- For space-efficient construction of the k-d tree, it is better NOT to copy the point lists into two half lists for the sub-problems in the recursion. Instead, pass the whole array to the sub-problems with the lower and higher index values that the recursive call is responsible for.
- "*#pragma omp task*" directive is useful for parallel recursive implementations. You may find resources in the internet about this.
- You should see speed-ups on both construction of k-d tree and processing the range queries for sufficiently large inputs.

7 Report

You should write a short report (not more than 5-6 pages). **Reports should be in .pdf format**, submissions with wrong format will get 0. Your report should include:

- *System Specifications.* Put the following at the start of your report:
  - Operating System:
  - CPU:
  - Number of Logical Cores[4]:
  - RAM:
  - HDD/SSD:
  - Compiler:
- *Implementation Details.* Briefly explain your implementations. Explain what data structures you used, which OpenMP directives you utilized etc.
- *Analysis.* Try to answers questions such as; What are your expected results? What is the percentage of the program that can be/is parallelized? Don't forget to mention the *perf* tool output in your discussions.

[3]-fno-omit-frame-pointer parameter is important!

[4]`lscpu` command on Linux systems gives this information in the output at the line starting with "CPU(s):".

- *Experiments.* Plot graphs for showing performance of your programs. Experiment with various inputs by changing dimension, number of points, and number of queries. You should give the speed-up values for each experiment. For generating input files, we share two Python scripts that you may use.
- *Discussion.* Discuss your results that answers questions such as; Does the parallel implementation improve the performance? What kind of input does it improve? If it does not improve, what might be the reasons?

8 Grading

- Serial Implementation: 20 points
- Parallel Implementation: 40 points
- *Perf* Tool Outputs for at least 3 Different Inputs [5]: 10 points
- Report: 30 points

9 Submission

Submit a single zip file **(yourname_lastname_p3.zip)** that includes:

- Your implementation with source files with the following naming convention and any additional source files:
  - `kdtree_serial.c` or `kdtree_serial.cpp`
  - `kdtree_parallel.c` or `kdtree_parallel.cpp`
- A Makefile that generates the following 2 executables: [6]
  - `kdtree_serial`
  - `kdtree_parallel`
- Your report
- *perf* or *gprof* output files in .txt format.

Submit to the respective assignment on Moodle.
No Late Submission Allowed!

References

[1] Thomas H Cormen et al. *Introduction to Algorithms.* MIT press, 2009.

[2] Franco P Preparata and Michael I Shamos. *Computational Geometry: An Introduction.* Springer Science & Business Media, 2012.

[5] Alternatively *gprof* results if you used it instead of *perf*

[6] We will execute the make command in the root directory of your project folder, the executables should be generated in this folder as well.