Second Assignment - FHPC Course
KD-tress Construction using parallel programming - MPI and OpenMP

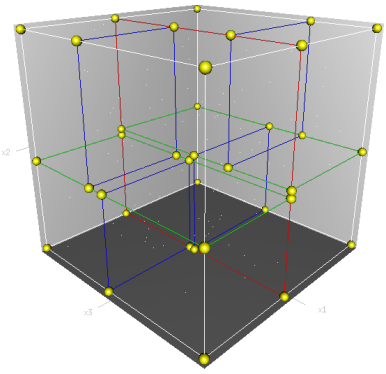
Elham Babaei

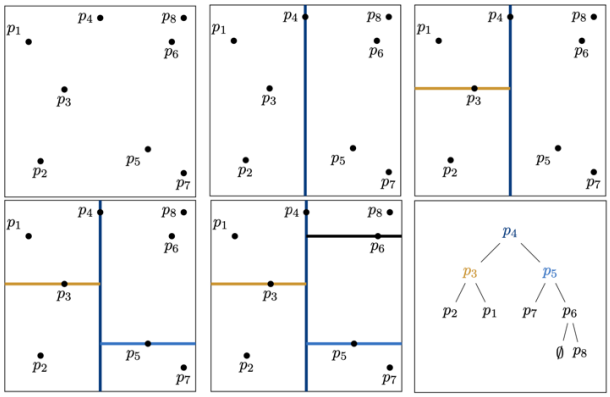
2021/2022

Content

Introduction

A k-d tree (short for k-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. The following figures show the partitioning in a 3d-tree and 2d-tree in 3-dimensional and 2-dimensional spaces respectively. The steps to create a 2d-tree on 8 datapoints and the result tree are shown on the right figure.





In this project, we are going to build a large 2d-tree in a 2-dimensional space of 10^8 datapoints. To create such a big 2d-tree we opt to use parallel programming in which the problem can be solved in a shorter amount of time using multiple processors.

Algorithm

To create a 2d-tree, we get the median of the dataset to split it into two left and right partitions; and we do the same recursively on each partition until we get the leaves of the tree. Since our data points are randomly generated from a uniform distribution, we simply switch between the 2 axises in each iteration based on Round Robin approach.

1. **Input** array of 2d nodes
2. **Output** subtrees
3. **Function** make_tree(datapoints_start, length, axis, ndim)
4. **If** (!length)
5. **return** 0
6. **End if**
7. **If** length >1
8. **median_point** ← **find_median**(datapoints_start, datapoints_start + length, axis)
9. **myaxis** ← round robin to change the axis

```
10.      median_point.leftTree ← make_tree(datapoints_start, median_point –
11.      datapoints_start, Myaxis, ndim)
12.      median_point.rightTree ← make_tree(datapoints_start + 1, datapoints_start +
13.      length – (median_point +1), myaxis, ndim)
14.   End if
15.   Return median_point which is a tree node
16. End function

17. Function find_median(datapoints_start, datapoints_end, myaxis)
18.   If datapoints_end <= datapoints_start
19.      return null
20.   End if
21.   If datapoints_end == datapoints_start +1
22.      return start
23.   End if
24.   median ← datapoints_start  + (datapoints_end - datapoints_start)/2
25.   Pivot ← median[myaxis]
26.   Function swap (median, datapoints_end – 1)
27.  Return median
28. End function
```

Implementation

The dataset is considered as an array of 2-dimensional data and datapoints are generated randomly from uniform distribution. As the first step to create a 2-d tree, we find the median point of the dataset based on one of the two directions. Then we split the data into two partitions left and right and we keep the median as the root of the tree. We continue doing the same on each left and right datasets; so we find other splitting points of the tree and since the dataset is coming from a uniform distribution and datapoints are monotonously distributed in the space, we switch the splitting direction based on Round Robin approach each time we create a new subtree. We continue this procedure until we get all the leaves of the tree.

To implement this process in parallel, we use a hybrid approach of MPI and OpenMP. The dataset is defined on processor 0 and distributed in equal-size chunks between all the processors. Thus, each processor has the same load of data and they create their own trees. Moreover, each processor creates an OpenMP parallel region and uses two OpenMP tasks to create left and right subtrees. We fixed the number of OpenMP threads equal to 2 for all the runs to study the speed-up of program.

The following image shows a simple run for which we have 16 datapoints in our dataset, running on 4 MPI processors and 2 OpenMP threads.

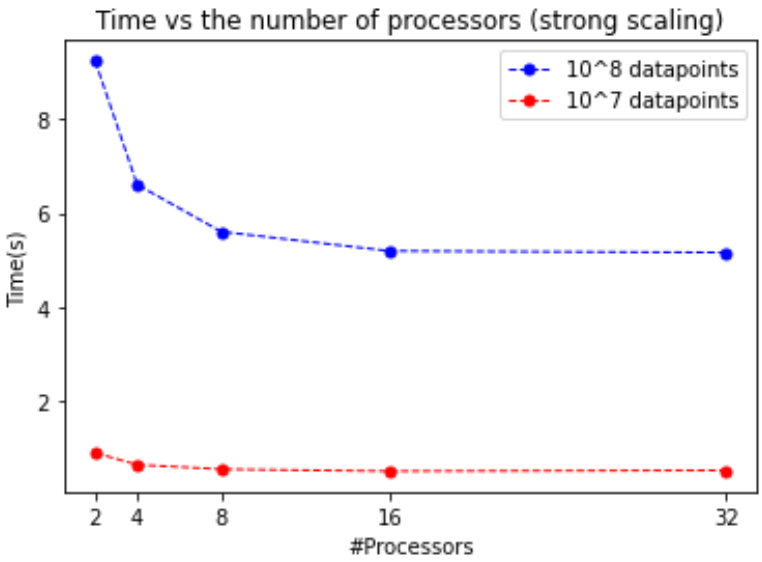
```
mpi processor 0, openmp thread 0 creating left tree
mpi processor 0, openmp thread 0 creating right tree
mpi processor 0, openmp thread 1 creating left tree
mpi processor 0, openmp thread 1 creating right tree
mpi processor 1, openmp thread 0 creating left tree
mpi processor 1, openmp thread 0 creating right tree
mpi processor 1, openmp thread 1 creating left tree
mpi processor 1, openmp thread 1 creating right tree
mpi processor 2, openmp thread 0 creating left tree
mpi processor 2, openmp thread 0 creating right tree
mpi processor 2, openmp thread 1 creating left tree
mpi processor 2, openmp thread 1 creating right tree
mpi processor 3, openmp thread 0 creating left tree
mpi processor 3, openmp thread 0 creating right tree
mpi processor 3, openmp thread 0 creating right tree
mpi processor 3, openmp thread 1 creating left tree
 # walltime on processor 0 : 0.00009290
 # walltime on processor 1 : 0.00010601
 # walltime on processor 2 : 0.00011438
 # walltime on processor 3 : 0.00009100
```

Performance model and scaling

In this section we study strong and weak scalability of the program. In order to measure strong scalability, we fixed the number of data points equal to 10^8 and 10^7 each time. The running time for different number of MPI processors is reported in the following tables and plot.

10^8 datapoints	
#Processors	**Time(S)**
2	9.24811097
4	6.6106786
8	5.60731258
16	5.19731976
32	5.16421028

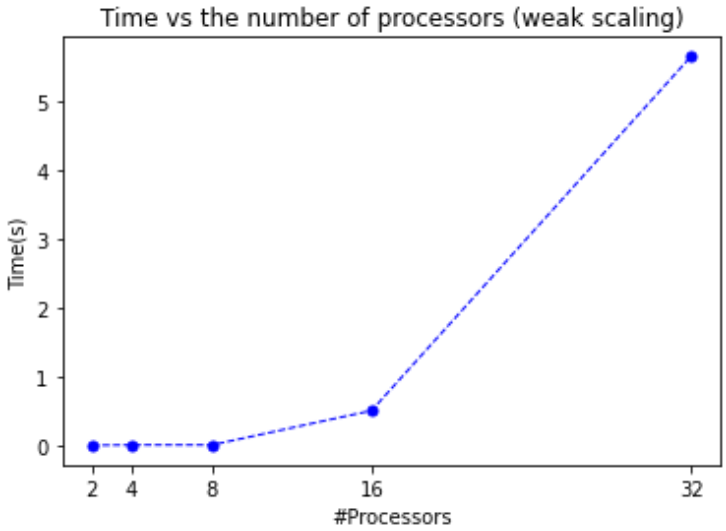
10^7 datapoints	
#Processors	**Time(S)**
2	0.89242187
4	0.64017095
8	0.54575861
16	0.5023067
32	0.51957803



We can see from the above plot that program doesn't scale well when we use larger number of processors, especially if the dataset is small. Therefore, the parallelization efficiency decreases as the number of resources increases. This is compatible with Amdahl's law.
This indicates that parallel programming should be done when it worth it and we are dealing with huge amount of data and a complicated program.

Weak scalability is studied in the below table and plot in which the size of the problem and the number of processors change together.

#Processors	Problem Size	Time (s)
2	10,000	0.00116712
4	100,000	0.00632658
8	1,000,000	0.00685021
16	10,000,000	0.50631095
32	100,000,000	5.65723818



The plot shows a significant jump in the running time for the case of 10^8 datapoints even if we are using 32 processors. So, the program always scales as the number of resources and the problem size increase; according to Gustafson's law**.**

Discussion

Gathering the created subtrees from each processor to create the final 2d-tree is not implemented here; the roots of already existing subtrees on each processor should be tracked and compared to do this. A better algorithm could be starting from the root processor, creating partitions, sending half of the data to the next processor, and doing this process in order until we catch all leaves. In both approaches, sorting the data could be a good way to increase the performance.