kd-tree construction and analysis with OpenMP and OpenMPI

Imola Fodor SM3500474
Foundations of High Performance Computing
University of Trieste

Deadline 28.02.2022

Contents

1 Introdution

A K Dimensional tree (or k-d tree) is a tree data structure that is used to represent points with more than one property in a k-dimensional space. It is a convenient way to organize points by several criteria at once and it provides eg. a convenient way to search, cluster points by their overall similarity.
In this work an effective and efficient *todo add time complexity* way to build and parallelize such a tree is presented.

2 Algorithm

The construction of the tree is done by:

- Finding median/pivot by median of medians of the input array. Each section of constant length is sorted by insertion sort.
- 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. Complexity
- Return root node.

The time complexity of the divide and conquer algorithm is O(nlogn) since the pivot is chosen in O(nlogn) time. If the partitioning would be done in a non optimal way (choosing as pivot the lowest, largest element of the array), the complexity would be O(n^2).

Algorithm 1 Build kD-tree

Input arrayOfNodes
Output treeRootNode

```
1: function BuildKDTree(startNode, length, axis, dim)

2:     if length = 0 then
3:         return 0
4:     end if

5:     myaxis ← round robin approach between 0 and 1

6:     medianNode ← MedianOfMedians(startNode, startNode + length
   -1, myaxis, len)

7:     medianNode.left ← BuildKDTree(startNode, medianNode -
   startNode, myaxis, dim)
8:     medianNode.right ← BuildKDTree(startNode, startNode +
   length - (medianNode + 1) , myaxis, dim)
9:     return treeNode
10: end function

11: function MedianOfMedians(startNode, endNode, myaxis, length)
12:     if length < 10 then
13:         InsertionSort(startNode, length, myaxis)
14:         median ← middleElement
15:     else
16:         subarrays ← ceiling(n/5)
17:         allocate array medians of length subarrays
18:         for i ← 1, subarrays do
19:             InsertionSort(startNode, length, myaxis)
20:             medians[i] ← middleElement
21:         end for

22:         if numSubarrays = high then
23:             median ← MedianOfMedians(medians, end, myaxis, length)
24:         end if
25:     end if
26:     return median
27: end function

28: procedure InsertionSort(startNode, length, axis)
29:      similar to the sorting of playing cards in hands
30: end procedure
```

3 Implementation

Herein the strategy used for OpenMP, OpenMP and their hybrid solution.

3.1 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 simplified, decorated Algorithm1 with the instruction read by OpenMP during compilation.

Algorithm 2 Build kD-tree w/ OpenMP

Input arrayOfNodes
Output treeRootNode

```
1: function MAIN
2:     #pragma omp parallel
3:     #pragma omp single nowait
4:     initialize random arrayOfNodes
5:     root ← BuildKDTree(arrayOfNodes, length, 0, 2)
6:     #pragma omp barrier
7:     printroot
8: end function
9: function BuildKDTree(startNode, length, axis, dim)

10:    if length = 0) then
11:        return 0
12:    end if

13:    myaxis ← round robin approach between 0 and 1

14:    medianNode ← MedianOfMedians(startNode, startNode + length
   -1, myaxis, len)

15:    #pragma omp task
16:    medianNode.left ← BuildKDTree(leftPoints, length, myaxis, dim)

17:    #pragma omp task
18:    medianNode.right ← BuildKDTree(rightPoints, length, myaxis,
   dim)
19:    return treeNode
20: end function

21: function MedianOfMedians(startNode, endNode, myaxis, length)
22:    if length < 10 then
23:        InsertionSort(startNode, length, myaxis)
24:        median ← middleElement
25:    else
26:        for i ← 1, subarrays do
27:            #pragma omp parallel for
28:            InsertionSort(startNode, length, myaxis)
29:            medians[i] ← middleElement
30:        end for
```



```
31:        median ← medians[middleElement]
32:    end if
33:    return median
34: end function
```

Continuing ...

```
35: procedure InsertionSort(startNode, length, axis)
36:     similar to the sorting of playing cards in hands
37: end procedure
```

3.2 Hybrid solution

The hybrid solution, other than leveraging parallelization within a node, leverages also the different nodes that reside in a cluster. To achieve this, point-to-point messaging needs to be designed along the NUMA node on disposal, since each node has its own memory space. Once a task receives it's message, each one computes the assignment on its portion of data using OpenMP threads, as under subsection OpenMP.

In order for each task to get it's portion of data, rank0, the master task, get's to find the nodes up until the level of the kdtree from where it is possible to assign the unique, fairly balanced, and only portions of data for each task (the left-most chunk being processed by rank 0).

Below the function that employs tasks, and the main function.

Algorithm 3 Build kD-tree w/ Hybrid

Input arrayOfNodes, numProcs, rank
Output treeRootNode

```
1: function MAIN
2:     #pragma omp parallel
3:     #pragma omp single nowait
4:     initialize random arrayOfNodes
5:     if rank = 0 then
6:         root ← FindFirstNodes(arrayOfNodes, length, 0, 2,
   depth=0,rank = -1)                                  ▷ rank is a pointer
7:     else
8:         MPI_Recv(length, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, ...)
9:         MPI_Recv(portion, length, MPI_BYTE, 0, 0, MPI_COMM_WORLD, ...)

10:        toSend ← BuildKDTree(leftPoints, length, myaxis, 2)
11:    end if
12: end function
13: function FindFirstNodes(startNode, length, axis, dim)

14:    myaxis ← round robin approach between 0 and 1

15:    if depth == log2(numProcs) then
16:        return 0
17:    end if

18:    medianNode ← MedianOfMedians(startNode, startNode + length
   -1, myaxis, len)

19:    #pragma omp task
20:    if depth == log2(numProcs) − 1 then
21:        rank ← rank + 1                          ▷ round robin fashion
22:        if rank = 0 then
23:            toSend ← BuildKDTree(leftPoints, leftLength, myaxis, 2)
24:        else
25:            MPI_Send(leftLength, 1, MPI_INT, rank, 2, MPI_COMM_WORLD)
26:            MPI_Send(leftPoints, leftLength, MPI_BYTE, rank, 0, MPI_COMM_WORLD)

27:        end if
28:        rank ← rank + 1                          ▷ round robin fashion
29:        MPI_Send(rightLength, 1, MPI_INT, rank, 2, MPI_COMM_WORLD)
30:        MPI_Send(rightPoints, rightLength, MPI_BYTE, rank, 0, MPI_COMM_WORLD)
```



```
31:    end if
```

Continuing ...

```
32:     depth ← depth + 1
33:     medianNode.left ← FindFirstNodes(leftPoints, length, myaxis, dim, depth, rank)
34:     #pragma omp task
35:     depth ← depth + 1
36:     medianNode.right ← FindFirstNodes(rightPoints, length, myaxis, dim, depth, rank)
37:     return treeNode
38: end function
```

4 Performance model and scaling

todo

5 Conclusion

OpenMPI strategy could have been to scatter insertion sort gather medians. Bug in code, sometimes some ranks do not finish, and sometimes there is an error regarding MPIRecv.