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 Introduction

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 a 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.
- Return root node.

The time complexity of the divide and conquer algorithm is O(nlogn) since the partition process always picks the middle element as pivot. The median of the medians is found in linear time, O(n).

Time complexity for partitioning n datapoints:

$$T(n) = 2T(n/2) + \theta(n) \tag{1}$$

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

function BUILDKDTREE(startNode, length, axis, dim)

    if *length* = 0 **then** ▷ base case
        return 0
    end if

    myaxis ← round robin approach between 0 and 1

    medianNode ← MEDIANOFMEDIANS(startNode, startNode + length -1, myaxis, len)

    medianNode.left ← BUILDKDTREE(startNode, medianNode - startNode, myaxis, dim)
    medianNode.right ← BUILDKDTREE(startNode, startNode + length - (medianNode + 1) , myaxis, dim)
    return treeNode
end function

function MEDIANOFMEDIANS(startNode, endNode, myaxis, length)
    if *length* < 10 **then** ▷ base case
        INSERTIONSORT($startNode, length, myaxis$)
        median ← *middleElement*
    else
        subarrays ← $ceiling(n/5)$
        allocate array medians of length subarrays
        for $i \leftarrow 1, subarrays$ **do**
            INSERTIONSORT($startNode, length, myaxis$)
            medians[i] ← *middleElement*
        end for

        if $numSubarrays = high$ **then**
            median ← MEDIANOFMEDIANS($medians, end, myaxis, length$)
        else
            INSERTIONSORT($medians, num\_subarrays, myaxis$)
            median ← $medians[middleElement]$
        end if
    end if
    return median
end function

procedure INSERTIONSORT(startNode, length, axis)
     similar to the sorting of playing cards in hands
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                                   ▷ base case
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                                   ▷ base case
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:        InsertionSort(medians, numsubarrays, myaxis)
32:        median ← medians[middleElement]
33:    end if
34:    return median
35: end function
```

Continuing ...

```
36: procedure InsertionSort(startNode, length, axis)
37:     similar to the sorting of playing cards in hands
38: 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, scaling up the solution. 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 rank, get's to find the tree 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). This approach was decided upon, to ensure load balancing and to easily reconstruct the tree, by sending the subtrees back to rank0.

MPI calls are made inside parallel regions, but there is no restriction when a certain thread receives the message from the master rank, hence the $MPI\_THREAD\_MULTIPLE$ paradigm is flagged.

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

11:        toSend ← BuildKDTree(portion, length, myaxis, 2)   ▷ ideally the head of the subtree would be sent back to rank 0, for full tree construct
12:    end if
13:    MPI_Finalize()
14: end function
15: function FindFirstNodes(startNode, length, axis, dim)

16:    myaxis ← round robin approach between 0 and 1

17:    if depth == log2(numProcs) then                    ▷ base case
18:        return 0
19:    end if

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

21:    #pragma omp task
22:    if depth == log2(numProcs) − 1 then
23:        rank ← rank + 1                                ▷ no round robin needed
24:        if rank = 0 then
25:            toSend ← BuildKDTree(leftPoints, leftLength, myaxis, 2)
26:        else
27:            MPI_Send(leftLength, 1, MPI_INT, rank, 2, MPI_COMM_WORLD)
28:            MPI_Send(leftPoints, leftLength, MPI_BYTE, rank, 0, MPI_COMM_WORLD)

29:        end if                          7
30:        rank ← rank + 1                                ▷ no round robin needed
31:        MPI_Send(rightLength, 1, MPI_INT, rank, 2, MPI_COMM_WORLD)
32:        MPI_Send(rightPoints, rightLength, MPI_BYTE, rank, 0, MPI_COMM_WORLD)

33:    end if
```

Continuing ...

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

4 Performance model and scaling

Measurements for speedup and efficiency graphs are used to acquire an indication of how well the implementation is performing in regard to some reference implementation. As reference, the corresponding serial execution of the code is used, on the same hardware, namely, for 10k datapoints, the execution time is 0.38s.

4.1 Hardware

- Device: Tesla V100-PCIE-32GB
- Multiprocessors: 48
- CUDA Cores/MP: 64
- Maximum number of threads per MP: 2048, per block 1028

4.2 Parallel speed-up

In order to compute the parallel speed-up of the implementation, the following formula is used:

$$S(P) = T(1)/T(P) \tag{2}$$

, where T(1) is the serial execution time, and T(P) is the parallel execution time of the same problem size, with P tasks.

4.3 Strong scaling

Parallel efficiency, also referred to as strong scaling, is calculated by:

$$E(P) = S(P)/P \tag{3}$$

When $S(P) = P$ it is considered a perfect speed-up. This in "real-life" is not an ultimate goal, since the code usually has parts that need to execute in a serial fashion. Ahmdal hence defined a way to model these more common, realistic implementations:

$$Sahm(P) = 1/(s + (p/N) \tag{4}$$

In case of the above described implementation, the purely serial part (s) is the initialization of the array of nodes. Furthermore, a show-stopper to a full parallelization is the part where the master rank computes the nodes up until a certain level of the tree.

Results 10k datapoints

Number of procs	2	4	8	16	32
Time [s]	0.39	0.42	0.42	0.42	x

No further analysis has been made, since no improvement in time has been noticed as the number of procs increased.

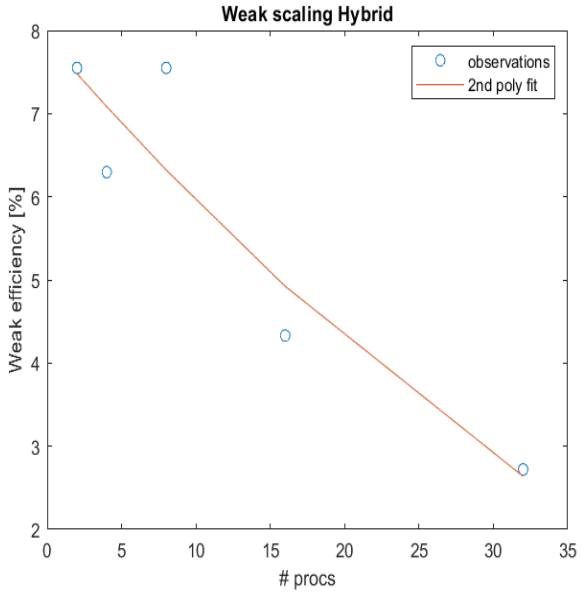
4.4 Weak scaling

Gustafson instead pointed out, that in practice the sizes of problems scale with the amount of available resources. This is called weak scaling, where the scaled speedup is calculated based on the amount of work done for a scaled problem size (in contrast to Amdahls law which focuses on fixed problem size). Gustafson proposed that with increased resources, the serial part remains the same, doesn't increase, even if the problem size increases.

The implementation of this work does not exactly comply with the latter assumption, since the more tasks we can employ, the master rank will have a bigger amount of levels (of the tree) to compute. Nevertheless, the weak scaling analysis is done, since the possible serial part is in any case, not utterly expensive, given the max amount of tasks we can employ.

$$Sgus(P) = s + pN \tag{5}$$

The weak efficiency can be calculated by eq.(3), changing the problem size with each change of resources. The problem size was increased by a factor of 10 (starting from $1k$), with each scale up in resources.



On *Figure*4. it can be noted that the efficiency starts at 8%, would confirm that the implementation is not very suitable for leveraging large resources.

4.5 Thread equilibrium for Hybrid solution

To understand better the optimal number of threads to be used for the hybrid approach, further analysis is needed. Using for the purpose the OpenMP function *omp_get_wtime*. It was noted that the program, to execute on $10k$ datapoints, took slightly lower time for 2 threads, $0.39s$ whileas for larger threads the time was $> 0.4s$.

5 Conclusion

The logic of the implementation in theory promised scaling with more tasks employed, but the timings extracted did not reflect the same.
The master rank slows the computation.

OpenMPI strategy could have been used to find the pivot element. In that case, rank 0 would have scattered the sections to the ranks, the ranks would have computed the insertion sort, to then send back their medians to rank 0, rank 0 gathering them.