Assignement 2 Report

Vaeria Insogna

Introduction

The aim of this project is to **build a 2 dimensional kd-tree** with the two main parallel computing paradigmas: OpenMP and MPI.

Kd-trees are a data structures presented originally by Friedman, Bentley and Finkel in 1977 to represent a set of k -dimensional data in order to make them efficiently searchable.

In order to simplify the task, the following two assumptions hold:

1. data set, and hence the related kd- tree, is immutable , i.e. you can neglect the insertion/deletion operations;
2. data points can be assumed to be homogeneously distributed in all the k dimensions.

A 2 dimensional kd-tree is a tree generated from *kpoints* (data points of 2 coordinates each) composed by *kdnodes* whose basic concept is to compare against one dimension at a time per level cycling cleverly through the dimensions so that to keep the tree balanced. At each kd-node, it bisects the data along the chosen dimension, creating a *left-subtree* and a *right-subtree* for which the following conditions hold:

- $\forall x \in subtree_{left}\ x_i < p_i$
- $\forall x \in subtree_{right}\ x_i > p_i$

where p is the pivot element (a kpoint) which creates the kdnode and splits the data set and i indicates the component of the i-th dimension of the kpoint. A node which has no children node (in my implementation left and right are NULL pointers) contains only a data point and is called a leaf.

Algorithm

In general, the choice of the pivotal point is clearly a critical step in the building of the tree since from that it descends how balanced the resulting tree will be. However, because of assumption 2) I decided to **pick the median element along the chosen dimension as pivot**.

The basic data structure for a kdnode of the kd-tree is the following:

```
struct kdnode{
    int axis; // the splitting dimension
    kpoint split; // the splitting element
    struct kdnode *left, *right; // the left and right sub-trees
}
```

with

```
struct kpoint{
    double coord[2];
};
```

The generated kd-tree that can be accessed by a kdnode-pointer with a **linked lists of kd-nodes**: the first address being the one that corrispondes to the first splitting kd-node at depth 0 of the kd-tree. For the MPI version see details in the suitable section.

The implementation of the algorithm used for finding the suitable splitting data point in a specific dimension, consists in the QuickSelect algorithm which finds the median and partially sort the array according to it.

QuickSelect

Like QuickSort, in an array of size N we pick as a pivot the first element of the array, then move the pivot element to its correct position and partition the surrounding array with a 3-ways partition (wich scans the array linearly time $O(n)$). The idea is, **not to do complete QuickSort that might be computationally intense (average case time complexity $O(nlogn)$), but stop at the point where pivot itself is k'th smallest element**: in our case $k = n/2$ is the median if n is odd or $k = n/2 + 1$ if N is even. The worst case time complexity of this method is $O(n^2)$, but **it works in $O(n)$ on average**.

In the implementation, if the number of left kpoints $n \leq 30$, then instead of the QuickSelect, a more time convenient algorithm for such few kpoints is performed: an **hybrid QuickSort**. It basically performs the InsertionSort ($O(n^2)$ in average) if the points are smaller than 10, otherwise it performs the QuickSort ($O(nlog(n))$ in average).

Splitting dimension

The choice of dimension for which is perform the splitting of the dataset could also be critical: if the data space is not sufficiently "squared"; otherwise, you may end up with nodes extremely elongated along one direction.

Once again, because of assumption 2) the simple choice to **round-robin through the dimensions** has been made.

Anyways, in my implementation I have taken this situation into account: by setting the variable `ROUND_ROBIN_AXIS=1`as true in file `kd_tree.c`, then the choice for the splitting dimension will be alternative between the two, otherwise it will depend on the amount of extention of the data in each dimension with the default threshold set as `EXTENT_DIFF_THRESHOLD=0.5`. With this algorithm in fact I can check the extent of the data domain along each direction and, if the extents are different by more than the threshold, it chooses as current splitting direction the one with the maximum extent (even if it was used in the previous iteration).

Since setting the `ROUND_ROBIN_AXIS=0` for our case of homogeneously random generated kpoints is not necessary and since it adds another $O(n)$ time complexity to the algorithm (which must scan all the array in the two dimensions to find max and min values for evaluating the extension), I have kept in my analysis a round-robin choice for the splitting dimension.

OpenMP

The algorithm for building the kd-tree is **recursive** and is done by means of **tasks**: the initial thread calls the function *build_kdtree* and it creates two tasks for each generation of the *left-subtree* and *right-subtree*, picked up by the available threads. The pseudocode for *build_kdtree* function is the one at reference 1.

Oss: the function GET_MEDIAN() returns the median via a pointer to the median kpoint with the QuickSelect algorithm. Moreover, this function takes charge of partioning the array with respect to the median.

MPI

Also for MPI, the algorithm for building the kd-tree is **recursive**. The work is shared among the processes and **it works only with number of processes equal to a power of two**.

The **master process randomly generates all the kpoints** and then it **calls the function** *build_kdtree* and **it delegates the work on the right branch to an idle process** (idle means not working process), whilst it remains to work on is *left-subtree*. The choice of the idle process is not trivial: see the Implementation section for reference. Moreover, the master process sends *right-subtree* data to the idle process which is waiting to collect them by function *prepare_build*.

Algorithm 1 OpenMP build_kdtree function called by $thread_0$

$N \leftarrow \#ofkpoints$
$ndim \leftarrow 2$
$axis \leftarrow -1$ ▷ Only starting axis is -1
$*kpoints \leftarrow \&\{p_1, p_2, ..., p_N\}$ ▷ Pointer to homo. rand. generated kpoints
$startIndex \leftarrow 0$ ▷ Starting index of subsection of kpoints
$finalIndex \leftarrow N - 1$ ▷ Final index of subsection of kpoints
function *BUILD_KDTREE($kpoints, ndim, axis, startIndex, finalIndex$) ▷ Ptr to new kdnode
  $N \leftarrow finalIndex - startIndex + 1$; ▷ # of kpoints in subsection
  if $N \geq 0$ **then**
    $myaxis \leftarrow (axis + 1) \bmod ndim$ ▷ Round-robin choice of axis
    $kdnode *$ `node` ▷ Ptr to new kdnode
    if $N = 1$ **then** ▷ Return a leaf
      $node \rightarrow axis = myaxis$
      $node \rightarrow split = kpoints[startIndex]$
      $node \rightarrow left = null$
      $node \rightarrow right = null$
    end if
    return `node`
    if $N = 2$ **then** ▷ Return a node prior to a leaf
      $node \rightarrow axis = myaxis$
      $node \rightarrow split = kpoints[startIndex]$
      $node \rightarrow left = null$
      $node \rightarrow right = BUILD\_KDTREE(kpoints, ndim, myaxis, finalIndex, finalIndex)$
    end if
    return `node`
    $kpoint *$ `median` $\leftarrow GET\_MEDIAN(kpoints, myaxis, N, startIndex, finalIndex)$
    $j \leftarrow startIndex + N/2$ ▷ Index of median in partitioned subsection
    $N\_left \leftarrow j - startIndex$
    $N\_right \leftarrow finalIndex - j$
    $node \rightarrow axis = myaxis$
    $node \rightarrow split =$ *`median`
    if $N\_left \geq 0$ **then**
      Task collected by $thread_i$**:**
      $node \rightarrow left = BUILD\_KDTREE(kpoints, ndim, myaxis, startIndex, j - 1)$
    end if
    if $N\_right \geq 0$ **then**
      Task collected by $thread_j$**:**
      $node \rightarrow right = BUILD\_KDTREE(kpoints, ndim, myaxis, j + 1, j + N\_right)$
    end if
    return `node`
  else
    throw error
  end if
end function

After all available processes are busy, the computation becomes serial.

At the end, each process has its own pointer to a linked list of kdnodes: in future improvements one should implement an additional algorithm to reduce all kdnodes spread among processes via low time complexity merging linked list procedure. The pseudocode for main core of *build_kdtree* function is the one at reference 2.

Algorithm 2 MPI core of build_kdtree and prepare_build functions

function $*$BUILD_KDTREE($kpoints, ndim, axis, startIndex, finalIndex$)
  $N \leftarrow finalIndex - startIndex + 1$;
  if $N \geq 0$ **then**
    $myaxis \leftarrow (axis + 1) \bmod ndim$
    $kdnode *$ `node`
    $kpoint *$ `median` $\leftarrow GET\_MEDIAN(kpoints, myaxis, N, startIndex, finalIndex)$
    $j \leftarrow startIndex + N/2$
    $N\_left \leftarrow j - startIndex$
    $N\_right \leftarrow finalIndex - j$
    $node \rightarrow axis = myaxis$
    $node \rightarrow split =$ `*median`
    if No idle process left then ▷ Start serial
      $node \rightarrow left = BUILD\_KDTREE\_SERIAL(kpoints, ndim, myaxis, startIndex, j - 1)$
      $node \rightarrow right = BUILD\_KDTREE\_SERIAL(kpoints, ndim, myaxis, j + 1, j + N\_right)$
    else
      if $N\_right \geq 0$ **then**
        $kpoint *$ `right_kpoints` $\leftarrow kpoints + j + 1$
        Send `right_kpoints` **to idle process**
        $node \rightarrow right = NULL$
      end if
      if $N\_left \geq 0$ **then**
        Continue building on left branch:
        $node \rightarrow left = BUILD\_KDTREE(kpoints, ndim, myaxis, startIndex, j - 1)$
      end if
    end if
    return `node`
  else
    throw error
  end if
end function
function PREPARE_BUILD
  Receive `right_kpoints` **from parent process**
  $kdnode *$ `sub_kdtree` $= BUILD\_KDTREE(right\_kpoints, ndim, myaxis, 0, N\_right - 1)$
  return `sub_kdtree`
end function

Implementation

I have implemented the program for building a 2-dimensional kd-tree in **C-language** (in serial and parallel ways) and compiled it with **GNU version 9.3.0** using optimization level `-O3`.

To make it easier, the program itself generates the data points (kpoints) on which to build the kd-tree: in compliance with assumption 2) they are by default `NDATAPOINTS=10^8` data points **homogeneously distributed** among [0,`MAX`], with `MAX=25`, **of type** `DOUBLE`. Moreover, at run time it's possible to change the number of kpoints by entering it as second argument (being the first argument the name of the executable).

If the variable `PRINT_TREE=1` is set at true in file `main.c`, then the program will print in the standard output

eventually the kd-nodes starting from depth 1 to `MAX_DEPTH=4` by default. For the MPI version, only the knodes of process with rank 0.

OpenMP

As stated in Algorithm section, the **kdtree is a shared pointer among the threads** and the first available thread starts building the tree:

```
#pragma omp parallel shared(data, ndim, kdtree, n)
    {
        #pragma omp single nowait
        // No syncro barrier at the end (!= single)
        {
            kdtree = build_kdtree(data, ndim, -1, 0, n-1);
        }
    }
```

Then inside $build_kdtree$ function, the left and right branch evaluation are assigned by tasks to the available other threads:

```
if(N_left > 0){
  #pragma omp task shared(ndim, points)
  //myaxis, startIndex, finalIndex are already firstprivate
  {
    new_node->left = build_kdtree( points, ndim, myaxis, startIndex, j - 1 );
  }
}
```

MPI

The major issue in the implementation of the MPI algorithm is to find an idle process to which delegate the computation of the right branch, when the number of processes is a power of two.

Since at each kdnode, the data set is partitioned in two sets with two different branches (right and left), the maximum parallel depth reachable with the given set of np processes (starting from 0) is:

$$depth_{max} = log_2(np) \tag{1}$$

Due to the approach used to parallelize the construction of the k-d tree, there in fact this maximum depth up to which every split is guaranteed to have at least one process ready to take on the right branch.

Thus, the approach I have chosen is to **use a strict pattern for the evaluation of the idle process rank based on the original process rank, the current depth of the kdtree and the number of processes involved**:

$$rank_{right} = rank_{current} + \frac{np}{2^{depth+1}} \tag{2}$$

Then if $log_2(np) \geq depth_{current}$ at some point in the iteration of the $build\_tree()$ function, the $rank_{right} = -1$ and the computation becomes serial in each process.

In this way, it is ensured that the rank given by the formula is free and ready to work.

The following is an example of the $rank_{right}$ evaluation with $np = 8$ processes (i.e. $depth_{max} = 3$).

```
--- level 0 ----
rank = 0 -> right_rank = 4
--- level 1 ----
rank = 0 -> right_rank = 2
rank = 4 -> right_rank = 6
```

```
--- level 2 ----
rank = 0 -> right_rank = 1
rank = 2 -> right_rank = 3
rank = 4 -> right_rank = 5
rank = 6 -> right_rank = 7
```

See section Discussion for future implementation of the choice of the right rank with a number of process not strictly as a power of two, but in the general case of $2^k + 1$ with $k \in N$.

In order to send/receive data from/to the right branch in the parallel phase, the following MPI routines has been used: `MPI_Send` and `MPI_Recv`.

Performance model and scaling

MPI case

In this section I am going to show the performance model of the **execution time for generating the kdtree in the MPI program** in order to check whether it resembles the actual measurements of the wall-clock time taken.

The MPI has two different phases:

- parallel phase in which each process:
  - partitions the dataset
  - finds the median
  - sends $right\_kpoints$ to $rank_{right}$ process via `MPI_Send`
  - compute recursively $build\_tree()$ on the left branch
- serial phase in which each process is independent:
  - partitions the dataset
  - finds the median
  - compute recursively $build\_tree()$ on the left branch
  - compute recursively $build\_tree()$ on the right branch

Parallel phase Here the majors time expenses are the QuickSelect algorithm and the one-to-one communication among processes for exchanging the data of the right branch.

The `MPI_Send`are actually two: one for sending the parameters needed for the recursive call of the and for acknowledge the size of the kpoints passed to the right branch, whilst the other is for storing those kpoints in the process memory.

```
// Send params for build_kdtree calling on right branch
int params[] = {N_right, ndim, myaxis, np_size, right_rank, next_depth, max_depth,
                surplus_np};
MPI_Send(params, 8, MPI_INT, right_rank, rank, comm);

// Send fraction of points to work on the right branch
MPI_Send(right_points, N_right * sizeof(kpoint), MPI_BYTE, right_rank, rank, comm);
```

To simplify the model, let's just consider the second `MPI_Send` and evaluate it:

$$T_{send}(n) = \lambda + \frac{ndim * n * double_{size}}{B} \tag{3}$$

with n the number of kpoints, λ $[s]$ the latency of the network, $ndim = 2$ the number of coordinates exchanged, $double_{size} = 8$ $[B]$ is the size of a double in Bytes and B $[B/s]$ is the bandwith of the network.

For the QuickSelect algorithm, instead, the complexity in time is, as already illustrated, $O(n)$ on average. For the sake of simplicity, let's ignore the fact that for $n \leq 30$ the QuickSort Hybrid approach is used.

This phase continues until there are no idle processes anymore, thus it is performed a number of $log_2(np)$ times, with np number of processes.

Thus, we can model the execution time:

$$T_{parallel}(n, np) = \sum_{i=0}^{log_2(np)} T_{send}(\frac{n}{2^{i+1}}) + c\dot{(}\frac{n}{2^{i+1}}) \tag{4}$$

with c a constant proper of this QuickSelect algorithm.

Serial phase Here each process works independently on its subtree, so let's evaluate the number of iterations. The algorithm stops when $n = 1$ and at each recursion the kpoints are split in half by the median, so the number of iterations are $log_2(n) - log_2(np)$. The execution model is:

$$T_{serial}(n, np) = \sum_{i=log_2(np)}^{log_2(n)} 2^{i-log_2(np)+1} + c\dot{(}\frac{n}{2^{i+1}}) \tag{5}$$

with again c the constant of the QuickSelect algorithm and $2^{i-log_2(np)+1}$ the term for the double partitioning operation at each subsequent split of the dataset.

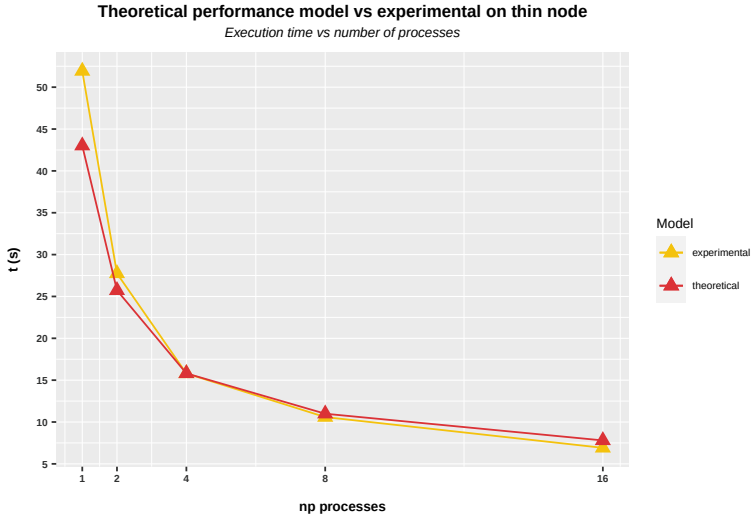
Total performance MPI The overall execution time is thus:

$$T_{exec}(n, np) = T_{serial}(n, np) + T_{parallel}(n, np) \tag{6}$$

Model evaluation To check the performance of the program, several measurements has been taken on an empty ORFEO thin node mapping the processes by socket.

The value of $\lambda = 0.485 \times 10^{-6}$ $[s]$ and $B = 5530.801 \times 10^{6}$ $[B/s]$ of the network has been measured by means of **IMB-MPI1 PingPong benchmark** across sockets.

Here I report a comparison between the mean of the 10 measurements with a fixed amount of data $n = 10^8$.



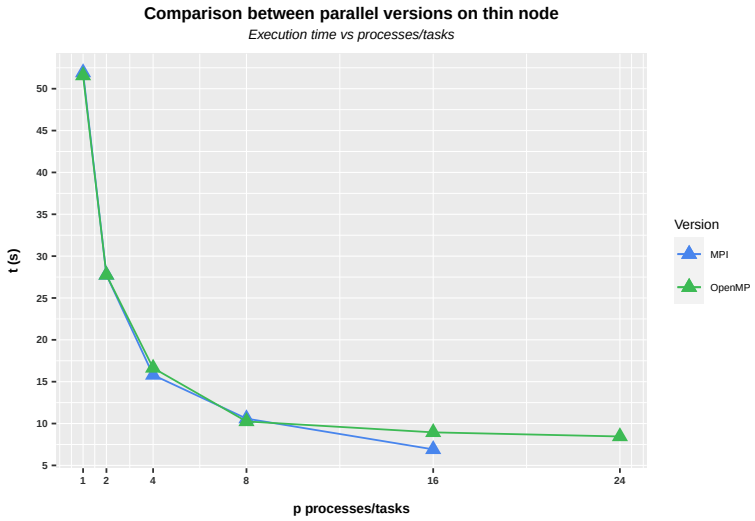
Hence, the overall complexity model seems to describe well the experimental results.

Strong scalability

A number of 10 measurements have been performed for each of the following parameters used:

- $n = 10^8$
- $p \in 1, 2, 4, 8, 16$

The following graph reports the mean among the 10 tests for each of the 5 configurations:



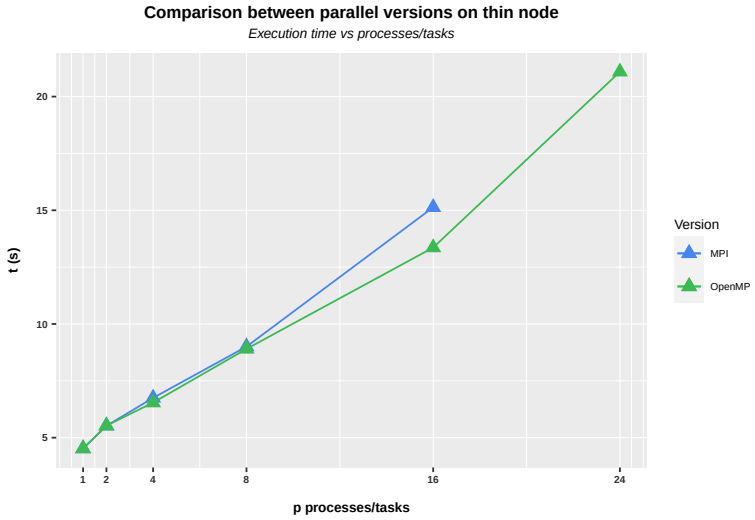
We can see that both MPI and OpenMP programs scale good, but OpenMp version has a slightly better speedup.

Week scalability

A number of 10 measurements have been performed for each of the following parameters used:

- $n = 10^8 \times p$
- $p \in 1, 2, 4, 8, 16$

The following graph reports the mean among the 10 tests for each of the 5 configurations:



We can see that both MPI and OpenMp do not scale weakly good, since linear increase of workload and processes/threads doesn't lead to constant time. The reason is due to the fact that only from a certain depth

of the tree there is no workload imbalance: at the beginning instead the first process/thread must partion an almost entire dataset which keeps growing with weak scalability.

Discussion

There are a few adjustments that could be made in order to perform better and there are several implementations to be improved.

False sharing in OpenMP In the OpenMP program there is a problem of false sharing and thus an increase workload to keep the cache coherence.

Tasks work with a specific and unique subsection of the kpoints in which they find the median element and partition the dataset.

Anyway, at a certain point, two tasks may work on adjacent subsection of the dataset which are close in the cache, and when swapping those close elements in the 3-ways partion they might fall into false sharing errors.

If we take into account that the cache lines are usually $64B$ long and that our data have 2 coordinates of doubles (aka $2 \times 8B$), then the number of kpoints storable in a cache line is 4.

The issue might be solved by putting an offset to each kpoint when generated and then take it into account when dealing with accessing/writing/swapping it.

This issue is anyway extremely costly in terms of memory expense, and considering that sorting operations of at most 4 elements are so small in number, we didn't use this fix.

MPI extension to any number of processes It should be possible to extent the amount of processes involved in the MPI program by simply distribute some workload of surplus non-working processes at the level of $depth_{max}$.

With surplus process is meant a process assigned to a level of the tree which does not guarantee that every splits has a process ready to take on the right branch. Only some splits have a surplus process, starting from the leftmost split in the tree.

So in the previous case of expected output for 10 processes (i.e. max_depth=3):

```
--- level 0 ----
rank = 0 -> right_rank = 4
--- level 1 ----
rank = 0 -> right_rank = 2
rank = 4 -> right_rank = 6
--- level 2 ----
rank = 0 -> right_rank = 1
rank = 2 -> right_rank = 3
rank = 4 -> right_rank = 5
rank = 6 -> right_rank = 7

Then surplus processes come to play:
rank = 0 -> right_rank = 8
rank = 1 -> right_rank = 9
rank = 2 -> right_rank = -1
rank = 3 -> right_rank = -1
```

This implementation has been carried out but not completed in my MPI version.

Hybrid approch It should be possible to add in the MPI version, an OpenMP section that make usage of tasks: instead of going serial from a certain point on, one could set a pragma parallel region inside each process.

Parallel QuickSelect in OpenMP Since we are working with tasks, it is possible to create sub tasks that perform the QuickSelect and 3-ways partion algorithms in parallel. In this way, one should really be careful when dealing with the data since serious false sharing errors might occur.

Code optimization As always, it should be possibile to optimize the code reducing the amount of overhead, setting loop unrolling etc..