Second Assignment FHPC course 2021/2022

version 0.94

Due date: 2022, Monday, February 28th @ 23:59

<<< note the change in the due date from v.0.91 >>>

Submission policy: create a folder on /fast/dssc/username/2021Assignment02 on ORFEO and push there the required files.

Required materials

In your folder you should provide the following files:

- report.pdf
  A document where you explain your algorithms and your implementation. You must provide a performance model of your code and a scalability study of the building-time as a function of the size of the input data set.
  The maximum size of the input data set (generate your own) must be at least 10^8 data points.
  Hint
  Tipically a report is expected to have this structure:
  - *Introduction* - a (very) brief and contained description of the problem
  - *Algorithm* - an abstract description of the algorithm that you have opted for to solve the problem
  - *Implementation* - an high-level description of the implementation (high level means that it is not necessary to discuss the small details of your code unless really needed to get the big picture of it)
  - *Performance model and scaling* - a discussion about the performance of your code: the performance model, which descends from your algorithm, and the strong and weak scaling that you have measured.
    How the scaling may be defined in case of hybrid codes?
    The scaling in that case refers to the scaling of MPI tasks with a fixed number of OpenMP threads per MPI task (obviously, the smaller is the latter the larger is the number of MPI tasks that you can use).
    As for the aforementioned fixed number of omp threads, you can either *(i)* choose a number, let's say 2, or *(ii)* you can perform a separate study running with a fixed number of MPI tasks while varying the number of threads per task. In this way, you will expose the behavior of your omp-ization and find the "optimal" task/thread decomposition (well, in general that may depend on the problem's size, actually, but do not worry about that).
    As for how to fix the number of omp threads to be used for the scaling of MPI tasks, you are free to opt for either one between *(i)* and *(ii)* mentioned above.
  - *Discussion* - any relevant topic you want to discuss; features and pitfalls, errores and defects of which you are aware, the motivation of design choices, proposals to further develop the code (not being satisfied because you know that you can write a better code may be a goos sign) and so on.
- kdtree.[c] [cc] [py]
- a script to compile your code. A Makefile will be appreciated

- note: due to lack of time, I could not settle the details of the kd-tree dump for you. Then do not consider this point as due anymore

  a file that contains the dump of your kd-tree from a data set that will be provided
  note: this last point is still TBD. I will add here shortly the expected format for dumping the kd-tree and the location of the data set

Build a *kd-tree* for 2-dimensional data

Kd-trees are a data structures presented originally by Friedman, Bentley and Finkel in 1977[1] to represent a set of *k*-dimensional data in order to make them efficiently searchable.

In spite of its age, or more likely thanks to it and to the large amount of research and improvements accumulated in time, *kd*-trees are still a good pragmatical choice to perform *knn* (k-nearest neighbours) operations in many cases.

In this assignment you are required to write a *parallel* code that builds a *kd*-tree for *k*=*2*. **You must implement both the MPI and the OpenMP version**.

In order to simplify the task, the following 2 assumptions hold:

- **[A1]** the data set, and hence the related *kd*-tree, can be assumed *immutable*, i.e. you can neglect the insertion/deletion operations;
- **[A2]** the data points can be assumed to be homogeneously distributed in all the *k* dimensions.

You find a formal introduction[2] to *kd*-trees in the uploaded paper "The k-d tree data structure and a proof for
neighborhood computation in expected logarithmic time", which contains also pseudo-code that guides you to

Some elements about the kd-trees

A *kd*-tree is a tree whose basic concept is to compare against 1 dimension at a time per level cycling cleverly through the dimensions so that to keep the tree balanced. The, at each iteration i, it bi-sects the data along the chosen dimension d, creating a "*left-*" and "*right-*" sub trees for which the 2 conditions $\forall x \in \text{sub-tree}, x_i < p_i$ and $\forall x \in \text{sub-tree}, x_i > p_i$ hold respectively, where $p \in D$ is a point in the data set and the sub-script i indicates the component of the $i-th$ dimension (in $2d$ you may visualize it better as the x and y cooordinates).

In general, the choice of the pivotal point p is clearly a critical step in the building of the tree since from that it descends how balanced the resulting tree will be. In fact, a large fraction of theoretical work on $kd-$trees is exactly about that.
However, here is where the assumption A2 above comes in to simplify this assignment; since you can assume that your data are homogeneously distributed in every dimension, a simple and good choice is to pick the *median* element along each dimension.
Then, the basic data structure for a node of your *kd*-tree should resemble to something like the following

```
#if !defined(DOUBLE_PRECISION)
#define float_t float
#else
#define float_t double
#endif
#define NDIM 2
struct kpoint float_t[NDIM];
struct kdnode {
   int axis;                        // the splitting dimension
   kpoint split;                    // the splitting element
   struct kdnode *left, *right;     // the left and right sub-trees
}
```

NOTE: the floating point size of the coordinates could be either float or double; in the code snippet here above you find the way to decide it at compile-time.

A node which has no children node (i.e. `left` and `right` are `NULL` pointers) contains only a data point and is called a *leaf*.

The choice of which is the splitting dimension at each iteration is also critical. A quite common basic choice is to round-robin through the dimensions, like

```
struct kdnode * build_kdtree( <current data set>, int ndim, int axis )
{
   // axis is the splitting dimension from the previous call
   // ( may be -1 for the first call)
   int myaxis = (axis+1)%ndim;
   struct kdnode *this_node = (struct kdnode*)malloc(sizeof(struct kdnode));
   this_node.axis = myaxis;

   ...;

   this_node.left = build_kdtree( <left_points>, ndim, myaxis);
   this_node.right = build_kdtree( <right_points>, ndim, myaxis);
   return this_node;
}
```

This strategy is good enough if the data space is sufficiently “squared”; otherwise, you may end up with nodes extremely elongated along one direction. Moreover, in the previous code snippet the check about the fact that at least 2 points are present in the chosen direction, which is mandatory, is absent.

Another possible and simple strategy is to pick the dimension of maximum extent. In other words, you may check the extent of the data domain along each direction and, if the extents are different by more than a threshold, you choose as current plsitting direction the one with the maximum extent (even if it was used in the previous iteration).
In this seconda case you treat “automatically” the case for strongly degenerated data distribution (in your case imagin a “stripe” distribution along either the x or the y axis).

All-in-all, a pseudo algorithm to build a *kd*-tree may resemplbe to something like the following:

```
struct kdnode * build_kdtree( kpoint *points, int N, int ndim, int axis )
   /*
    * points is a pointer to the relevant section of the data set;
    * N is the number of points to be considered, from points to points+N
    * ndim is the number of dimensions of the data points
```

```
     * axis is the dimension used previsously as the splitting one
     */
{
   if( N == 1 ) return a leaf with the point *points;

   struct kdnode *node;
   //
   // ...  here you should either allocate the memory for a new node
   // like in the classical linked-list, or implement something different
   // and more efficient in terms of tree-traversal
   //
   int myaxis = choose_splitting_dimension( points, ndim, axis);   // implement the choice for
                                                                   // the splitting dimension
   kpoint *mypoint = choose_splitting_point( points, myaxis);      // pick-up the splitting point

   struct kpoint *left_points, *right_points;
   //
   // ... here you individuate the left- and right- points
   //

   node -> axis = myaxis;
   node -> split = *splitting_point;    // here we save a data point, but it's up to youù
                                        // to opt to save a pointer, instead
   node -> left = build_kdtree( left_points, N_left, ndim, myaxis );
   node -> right = build_kdtree( right_points, N_right, ndim, myaxis );
   return node;
}
```

A critical points that is left to your implementation is how to "prepare" the data set *before* starting the tree construction. At every iteration you have to find *(i)* the extent along each direction, *(ii)* the splitting point (which is kind ok the "median", i.e. the one that lies in the middle of the distribution) and *(iii)* the left- and right- sub-sets of the current data set.

Here the assumption A1 may helps you simplifying the matter.
Just to give some possible strategy you may:

1. Leave the data untouched, with a first scan that finds the limits of the extent in each direction; hence at each iteration, dealing with *N* points, you may scan them linearly until you find the closest to the middle of the extent along the splitting dimension (you can do that because of assumption A2). The, you can use thaat point coordinate to partition the point set in two halves. And so on.
2. Pre-sort the data in each direction. You may sort the data set itself along one direction, using the quicksort or the merge sort routines provided); do you need another copy in order to be sorted in the other direction or you may think to something different ?
   In any case, having - in some way - the data sorted along each direction could greatly simplify many tasks.

Which one, between strategy 1 and 2 here above, is expected to be more conveniente in terms of the average number of operations expected? Please add some comment on this to the performance analysis that you will include in your report (*note: it is not required to implement both strategies*).

Parallelization

You are required to implement both an MPI and an OpenMP versions of the code. Alternatively, you could opt for a single hybrid MPI-OpenMP version.
The parallelization strategy may be pretty similar.
One of the simplest strategy, given assumptions A1 and A2, is that you may distribute the data among your processes/threads (let's call them "tasks" in the following, meaning either MPI processes or OpenMP threads).
For instance, if you have 2 tasks, you may determine the first splitting and hence distribute the data among the two workers. Each task may then proceed independently. The same strategy holds for a larger number of tasks in an abvisou way. If you choose this approach, **you can assume to use a number of tasks that is a power of 2**.

---

1. Jerome Harold Friedman, Jon Louis Bentley, and Raphael Ari Finkel. "An algorithm for finding best matches in logarithmic expected time". In: ACM Transactions on Mathematical Software (TOMS) 3.3 (1977), pp. 209–226. See the materials added in the sub-folder `materials/` in the assignment folder. ↩

2. the wikipedia page for the kd-trees is also a good starting point for your needs. ↩