Exercise 1

Parallel & Distributed Computer Systems

Oct 14, 2019

We will implement step-by-step a shared memory vantage point tree construction, given a data set of d-dimensional points X. The structure is used to perform the k nearest neighbor search (kNN) of the points on the vantage-point tree as introduced in the publication:

> *Peter N Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, In Fourth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, volume 93, pages 311–321, 1993.*

The algorithm is based on a very simple idea: select a point as the vantage point, compute the distance of every point to the vantage point and split the points according to the median distance.

```
clear

n = 100000; d = 2; % works for any d

X = rand(n,d);

% assume vantage point is the last one
% get squares of distances from it
d = sqrt( sum((X(1:n-1,:) - X(n,:)).^2,2) );
% and find the median distance
medianDistance = median(d);

% plot them to confirm
clf; hold on; axis equal

plot(X(d <= medianDistance,1), X(d <= medianDistance,2), 'r.')
plot(X(d >  medianDistance,1), X(d >  medianDistance,2), 'b.')
plot(X(n,1), X(n,2), 'bo') % vantage point
```

And then repeat the same on the inner and outer partitions until no point is left, to build a balanced binary tree.

```
function T = vpTree(X)
% function T = vpTree(X)
% computes the vantage point tree structure with
%   T.vp : the vantage point
%   T.md : the median distance of the vantage point to the other points
%   T.idx : the index of the vantage point in the original set
%   T.inner and T.outer : vantage-point subtrees
% of a set of n points in d dimensions with coordinates X(1:n,1:d)
%
  T = vpt(X, 1:size(X,1));

  function T = vpt(X, idx)

    n = size(X,1); % number of points
    if n == 0
      T = [];
    else
      T.vp = X(n,:);
```

```
      T.idx = idx(n);

      d = sqrt( sum((X(1:n-1,:) - X(n,:)).^2,2) );

      medianDistance = median(d);
      T.md = medianDistance;

      % split and recurse
      inner = d <= medianDistance;
      T.inner = vpt(X( inner,:), idx( inner));
      T.outer = vpt(X(~inner,:), idx(~inner));
    end
  end
end
```

You do not need to follow the MATLAB implementation shown here, but make sure you understand all of the above and design tests to confirm correctness so that we can rewrite it in C and then parallelize it.

0. Implement the vantage-point tree in C (2 points)

Pay attention your implementation to be correct and efficient. Decide what needs to be copied and what can be done in place. Find the median distance using Quickselect.

1. Parallelize your implementation using `pthreads` (3 points)

There are two things you can do in parallel

1. compute the distances in parallel
2. compute the inner set in parallel with the outer set

We can also compute the median in parallel, using the exact same techniques we discuss here, but it won't be as critical so you can skip it. Read about `pthreads` here.

2. Threshold the parallel calls (1 point)

Assigning too little work to be done in parallel will actually slow down your computations. Modify your parallel implementation to switch to call the sequential code after a certain threshold size. Also restrict the maximum number of live threads to an upper bound. Experiment to identify the optimal threshold value and maximum number of threads for your implementation and hardware.

3. Reimplement version #2 using `Cilk` and `OpenMP` (4 points)

Life is actually much easier when one uses high-level expressions of parallelism. Read about `Cilk` and `OpenMP` here and here. Modify your version 0, with annotations to reimplement code version 2 in `Cilk` and `OpenMP`. These compilers generate code very similar to what you have prepared by hand!

What to submit

- A 3-page report in PDF format (any pages after the 3rd one will not be taken into account). Report execution time of your implementations with respect to the number of data points n and the number of dimensions d.
- Upload the source code on GitHub, BitBucket, Dropbox, Google Drive, etc. and add a link in your report.

Your code must implement the accessors defined in the following header.

```c
#ifndef VPTREE_H
#define VPTREE_H

// type definition of vptree

// ========== LIST OF ACCESSORS

//! Build vantage-point tree given input dataset X
/*!
  \param X      Input data points, stored as [n-by-d] array
  \param n      Number of data points (rows of X)
  \param d      Number of dimensions (columns of X)
  \return The vantage-point tree
*/
vptree * buildvp(double *X, int n, int d);

//! Return vantage-point subtree with points inside radius
/*!
  \param node   A vantage-point tree
  \return The vantage-point subtree
*/
vptree * getInner(vptree * T);

//! Return vantage-point subtree with points outside radius
/*!
  \param node   A vantage-point tree
  \return The vantage-point subtree
*/
vptree * getOuter(vptree * T);

//! Return median of distances to vantage point
/*!
  \param node   A vantage-point tree
  \return The median distance
*/
double getMD(vptree * T);

//! Return the coordinates of the vantage point
/*!
  \param node   A vantage-point tree
  \return The coordinates [d-dimensional vector]
*/
double * getVP(vptree * T);

//! Return the index of the vantage point
/*!
  \param node   A vantage-point tree
  \return The index to the input vector of data points
*/
int getIDX(vptree * T);

#endif
```

Implement each version in a different source file `.c` and create a makefile with a rule `make lib` that generates the

following libraries

- `vptree_sequential.a`
- `vptree_pthreads.a`
- `vptree_openmp.a`
- `vptree_cilk.a`

The accessors must be defined for each version. You can check the Fibonacci example under Course Material > Examples for a `Makefile` and a project structure that follows similar guidelines.

Upload a `.tar.gz` compressed file of your project and add the link in the CodeRunner quiz to test the validity of your implementations. The project structure inside the `tar.gz` file should be the following

```
vptree/
|-- Makefile
|-- lib/
|-- src/
|-- inc/
```

The command `make lib` should generate the four libraries `.a` inside the `lib` directory under `vptree`. The `vptree.h` header with accessors defined above, should be found under `inc` directory.

DO NOT overuse the validation system in elearning. Please validate locally, and only upload when certain your implementation is valid to receive the grade.

Examples of `Makefile` usage with `pthreads`, `OpenMP`, and `Cilk` are available under Course Material > Examples.

Groups of maximum 2 person are allowed. Both students must upload their report on the submission system.

Afternotes

Observe that the time for the construction of the vantage point tree grows linearly with the number of points, and very slowly with the increase of the dimensions!

The vantage point tree is used to search for nearest points to given query points. The idea is that the tree will allow the pruning of most points that are too far away to be considered for the k neighbors at an average cost of $\log n$. In the search we descend first in the subtree that the query point belongs to, to find the neighbors and search the other subtree only when the k^th neighbor's distance intersects the other subspace. Implementing the search for a single query point in parallel efficiently is hard.

In all kNN, we use the same set of nodes to make and query the tree. The obvious parallelism is to split the queries among the threads.
In a truly high performance implementation, you won't continue the recursion to a single point per leaf, but to a number of points that will be determined by the capability of the hardware and the quality of your implementation, just like the threshold from parallel to the sequential implementation. The sets of points per leaf can be processed in bulk, making better use of the vectorizing hardware and resulting to mostly better guesses for the k neighbors, thus achieving good pruning of the tree.