Vantage Point Tree

Ιακωβίδης Ιωάννης Αντονακούδης Σωτήριος

29 Δεκεμβρίου 2018

1 Introduction

The problem of finding the k-nearest neighbors of a set of points in space is of obvious importance to many fields. As the points can represent anything. A brute force solution to the problem would be finding the distance of the quire to each point from the set and then sort the resulting array. But this is time consuming and in a non random data set the points tend to form clusters and this doesn't use the structure of the problem at all.

The vantage point tree is a data structure that stores the points so that the search for every new quire has expected complexity $O(n \lg n)$. The construction of the tree is made by choosing the median of distances from a random pivot point and then splitting the array to the points that are closer than the median and the points that are farther than the median as left and right child respectively. So for each point we have a circle with radius the median distance and all the points that lie inside the circle are in the left subtree while all the points that lie outside the circle are in the right subtree. And the Search algorithm is of the form

Algorithm 1: Search(Tree,Point,heap,tau)

```
   Result: k-nearest neighbors
 1 update(tau,heap);
 2 if $|Tree.point - Point| < radius$ then
 3     Search(Tree.left,Point,heap,tau);
 4     if $tau \geq radius - |Tree.point - Point|$ then
 5         Search(Tree.right,Point,heap,tau);
 6 else
 7     Search(Tree.right,Point,heap,tau);
 8     if $tau \leq |Tree.point - Point| - radius$ then
 9         Search(Tree.left,Point,heap,tau);
10 end
```

The heap maintains the k closest points we have found yet and tau is the distance from the farthest one of those.

We made an MPI implementation of the Vantage point construction and search function that can run on a cluster with further speed up and can handle big sets of data.

Let the number of processes be p and the size of the set is N. p and N are chosen to be powers of 2 for easier implementation

2 Construction of the Tree

2.1 Structure Of The Tree

We assign to each process $\frac{N}{p}$ points in an array pointsArray[$\frac{N}{p}$]. And we construct the tree so that all of the processes have a tree common of p levels that and each process has it's own local tree of lg(N/p) levels. And in the end we attach the local tree to a leaf of the common tree so that the root of the local tree is the processId node in the p'th level. Each node of the tree contains the coordinates of a point, the radius of the circle , two pointers for the left and right child and also an identifier boolean that says if the node is a leaf or not.

2.2 Common Tree

In the main function. We construct this tree form the root level by level. From the starting size N until the size becomes N/p, we are left with the points of a single process ,we split the set in half via computing the median with all processes collectively and constructing a node from a random pivot point. After we swap parts of the arrays of the processes so that half the processes have points points inside the circle and half outside so that in the next level the each of the two sets of processes continuous for the right and left subtree.

```
for(depth=0;depth<q;depth++)
  {
    Medians(processId ,noProcesses ,partLength ,pointsArray ,
    vantagepoints ,size ,depth ,dimension ,medianArray);

        lvl=(struct Node*)malloc((1<<depth)*2*sizeof(struct Node));
        for(int i=0;i<(1<<depth);i++){
          dad[i].vpoint=(float *)malloc(dimension*sizeof(float));
          for(int j=0;j<dimension;j++)
            {dad[i].vpoint[j]=vantagepoints[i*dimension+j];}
          dad[i].leaf=false;
          dad[i].radius=medianArray[i];
          dad[i].left=&lvl[i*2];
          dad[i].right=&lvl[i*2+1];
          if(depth==q-1){
            dad[i].right=NULL;
            dad[i].left=NULL;
            if(i==floor(processId/2)){
            if(processId%2==0){
            temp=&dad[i].left;
            }else{
            temp=&dad[i].right;
          }}
        }
      }
      dad=lvl;

  }
```

The Medians function is used so that all the processes have 2^{depth} medians , form the right subtrees in mediansArray, for radius and also random the vantage points of the set in each level. Also does the swap of the subarrays. The temp variable is used so that we have the place to put the local tree. We have put NULL to the leafs of the common tree as identifiers for the search function (if it tries to be called to the NULL node it should be called in tree of an other process).

2.3 Local Tree

After the construction of the common tree its process creates is own Local Tree and attaches its root to the common tree.

```
localroot=createtreelocal(pointsArray, partLength, dimension,processId);
*temp=localroot;
```

The function createtreelocal returns the local tree. By choosing a random point computing the median distance with the selection function and recursing to the left and right half of the array for it's children.

3 Serial Search

The serial search in its processes tree is implemented by the Search function.

```
void Search2(struct Node* tree,float* point,int k,
int dimension,std::priority_queue<HeapItem>*heap,int pros,int* should)
```

Which uses the same algorithm described above and stores on the heap the k-nearest neighbors to point in the process Tree.

But there are some differences with a single process implementation. First the way we split the array when we create the tree allows for nodes to store the same point. But all of the points of it's process are in exactly one leaf. So we update the heap and tau (farthest distance for a point in the heap) only in the leafs of the tree. This will slow down the algorith untill we fil the heap with k elements because we will need to traverse more nodes. Second for each quire point we have to know if we need to also search tree of the other processes. For this we check with the use of the NULL pointers that are as children of the leafs of the common tree and we update the should array so that it's i is 1 or 0 if the point has to be searched by the i'th process or not. The argument pros is used for computing the process id that we need to update should for.

4 MPI-Search

Finally to Search the whole set we need to make the right process communications if needed. We implement the search between the points of the starting set but give a new set of quires same stategy can be implemented. The Communications are done in groups of 2^i consecutive id processes for $i = 1, \dots, \lg p$ That way implement all the communications and we start by making the pairing up processes of closer process number first. This is crucial because in that way we are more likely to update the heap and the should array so that we can trim communication and searches later. In each group the communications are done in a cyclic manner so that each process from the first half-group interactes with all of the processes of the right half-group.

```
            for(int level=q-1;level>=0;level--){
               int groupsize = ( noProcesses / (1<<level) ) ;
               int color = processId / groupsize ;
               MPI_Comm group;
               MPI_Comm_split(MPI_COMM_WORLD, color , processId , &group);
               int local_rank , local_size;
               MPI_Comm_rank(group, &local_rank);
               MPI_Comm_size(group, &local_size);
               if(local_rank<groupsize/2){
               for(int i=0;i<groupsize/2;i++){
                  int target=groupsize/2+(local_rank+i)%(groupsize/2);
                  //Communicate:Send,Recieve and Search targets points}
             }else{for(int i=0;i<groupsize/2;i++){
                  int target=(local_rank-i)%(groupsize/2);
                  //Communicate:Send,Recieve and Search targets points}}}
```

Each pair of processes exchange the heaps of all the points in the form of arrays ,heapdist and heapPoints , and also the should and processflag arrays that are used to skip communications and searches if possible. Processflag[i]=0 in process j if and only if the process i does not need to search any point of j. So we fist exchange processflags and make every communication that is needed, by searching in the receiving processes tree we update the heap, processesflag and should arrays and then send it back so that it's maintained.

We started we with creating the findNeighbors function we finds the k closest points to a given point from each process. After searching its local tree each process calls this function which hadles the communactions between the processes as described above. Calling this function for all the points would be slow since it would require a very large number of small data packets so we wrote the code to search for all the points at the same time.

5 Implementation Details

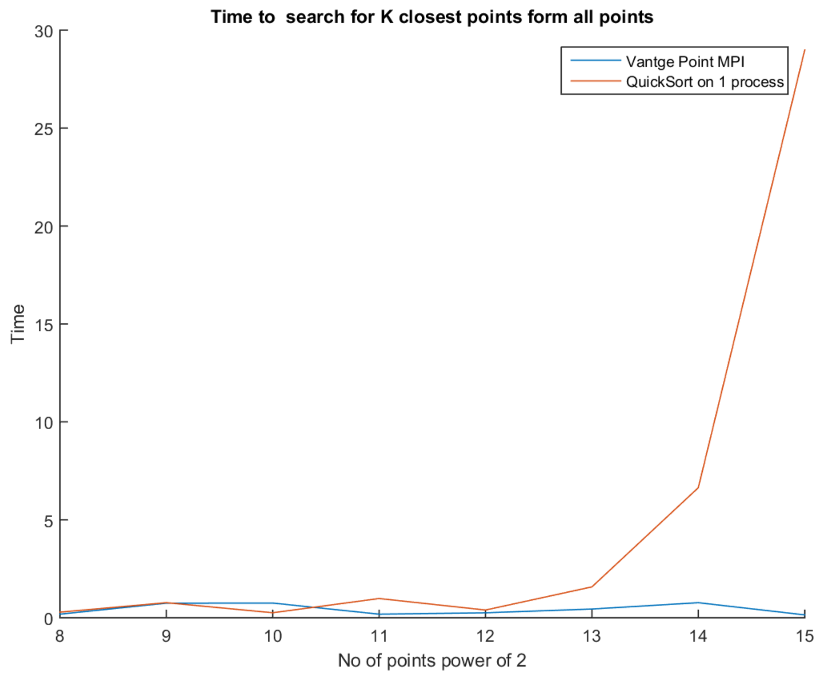
5.1 Medians Function

In order to find the median distance from the quire point and split the array we use the master part and slave part functions of the MpiFindMedian code. During the construction of the common tree the median function splits the mpi communicator world in subgroups of NoProsecces/2^{depth} . Each group represents a node of common tree and chooses its own vantage point which all the processes of the group use to find the distances from all their points. Then we call the master and slave part modified with the group as an mpi communicator type argument so they return the median only from the element of each group. The master of each group then collects all the points and splits them based on their distance so the first half processes get the points closest to the vantage point and the second half get the farthest. The function saves the location of the vantage points and the median distances of each subgroup to two arrays in all of the processes which we can use to create the vantage point tree.

6 Performance and Validity

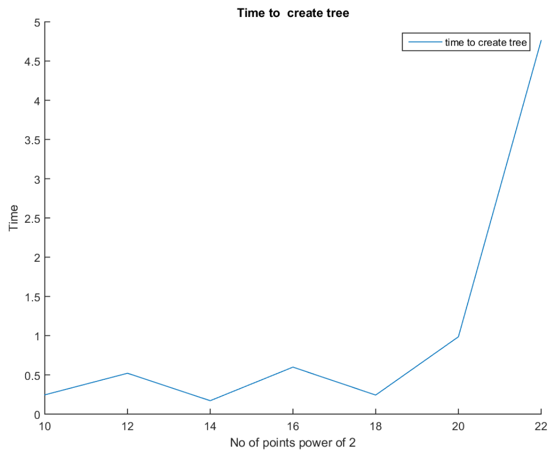
Concerning the Validity of the implementation we cross reference the results of the MPI-search with quicksort of the distances for every point of the set and print message if the results don't match. This not only asserts the correctness of the algorithm for a particular distance of the problem but also gives us a measure of the performance compared to a trivial solution of the problem.

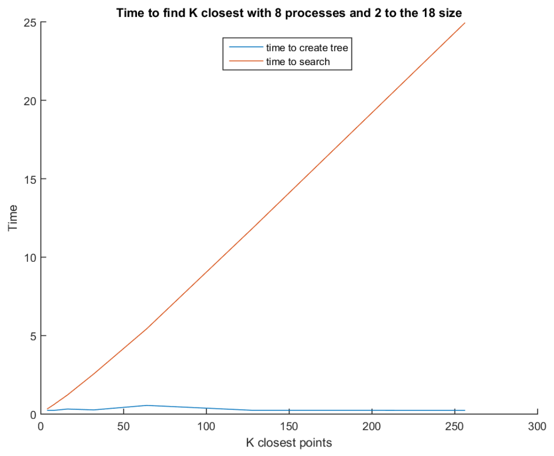
The following trials of the code were done in the diades system provided by the course.



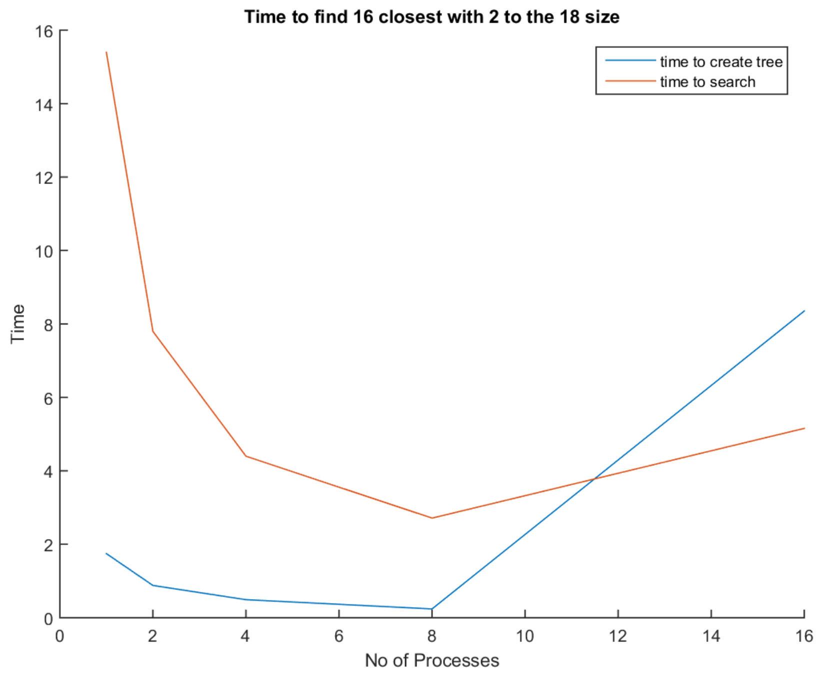
From the constructed graph we conclude that 2^{12} is the threshold that before the Vp-MPI implementation and the quicksort are comparable and after that we have exponential speed up.

But compared with an algorithm like quicksort the Vantage-Point-Tree search need it's data to be preprocessed (Tree Creation)

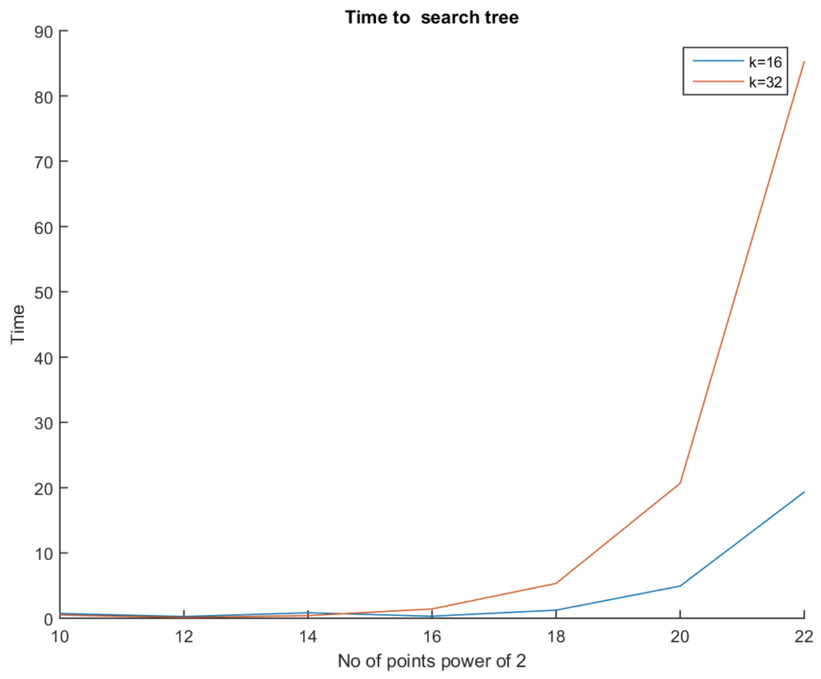




From the above graphs we can see data about the performance of the tree creation and also how it compares with the actual search. So we can conclude that the creation process takes much less time than the actual search and this gap gets wider if we ask greater number of neighbors. Thus making the implementation accessible especially ask for large set of neighbors.



Varing the number of processes we see that we get our pick performance for 8 processes both in creating the tree and searching it, understandable because we run our tests in an 8 core system. Also we see that the get a 2.66 in searching and a 5.6 in tree-creation speed up comparing 8 and 2 process executions. After 8 processes we have no speed up and the overhead of creating and managing the processes slows down our implementation in this system especially the creation.



Finally testing the scalability of the search (for 16 and 32 neighbors) we find that the time scales quadratically with the size of the problem and also we see k's effect in the scalability. W have time greater than 20 seconds for problem size greater than 2^{20} for k=32 and 2^{22} for k=16.

Hellas grid tests Problem size 2 to the power of 20 K=16			
No of Nodes	No of cores	Time to create tree	Time to search tree
1	8	1.572876 sec	5.566918 sec
2	16	1.90298 sec	10.616262 sec
4	32	1.329196	13.491008

7 Performace on Hellas Grid

After testing on a single machine the code was tested on the Hellas grid cluster with 1,2 and 4 nodes of 8 cores each. In this enviroment we observe slower performace since the communication between the nodes is much slower compared to the communication between processes on a single machine. With 2 nodes the penalty is very large as we get 2 times slower performace but with 4 nodes the benefits of parallel work are clearer as the performance is similart to 2 nodes. In order to reduce the penalty of distrubuted nodes, the code needs to optimize the communications in such way so to send only the nessecary data.

8 Code

CodeLink