Computational Capstone ORB

Gabriel Toban, Khem Poudel, Toheeb Biala

Paralle Pros

April 2019

Contents

1 Introduction

This project is intended to create an orthogonal recursive bisection (ORB) of the data set along N-dimensions with serial and parallel sort. The ORB is implemented for a parallel spatial binary tree that searches 10 billion points to find how many points are within 3 search radii of 20 million targets respectively. There are 3 main parts of this project: building a binary tree, sorting across ranks, and searching/counting the points. The binary tree is separated into 2 parts: a global tree and a local tree.

The global tree effectively finds the rank that a specified point exists on which are ordered according to the 3 dimensional spatial coordinates of the points that exists within it's upper and lower bounds. The global tree only exist on rank 0 (which we shall henceforth call the master rank). All the nodes in the global tree are a special structure (described later) referred to as global nodes or g nodes (gnodes). The node at the top of the global tree is referred to as the "global head". The leaves of the global tree are the head nodes of the local tree.

The local tree effectively finds the number of points that are within a radius to a target. Each rank stores it's own local tree. All nodes in the local tree are typical node structures (described later) in a binary tree referred to as local nodes or simply nodes. The node at the top of the local tree is referred to as the "local head". Even though the leaves of the global tree are equal to the heads of the local trees, they are stored in different places and use different structures. The leaves of the local tree store a pointer to the actual point they represent.

Sorting across ranks is completed through a bucketing algorithm. Each rank sorts it's elements in the direction of the dimension with the largest variance (largestDim or colIndex). There are always the same number of buckets as there are ranks (global number of ranks or *global_num_ranks*). Buckets are created by evenly dividing the points on each rank into equally sized buckets with the "right most" bucket holding the remainder elements. The boundaries or lower limits of those buckets across all ranks are sorted across all ranks. The lower limit positions, li, are chosen to be the global lower limit (Li) such that li evenly divides into the number of ranks. The global lower limits are adjusted until each rank contains an equal number of elements with the last rank containing the remainder elements. In summary, the buckets on each rank are used to equally divide the sorted data across all ranks 1 global lower limit at a time.

Counting the points is completed using a master node. Rank 0 searches the global tree for which targets should be assigned to which ranks. Rank 0 assigns those targets to each rank. Each rank then searches it's local tree for each target and each radius. Rank 0 then sums the number of points across all ranks for each target and radius.

This report discusses these main parts in the order they exists in the code using conceptual explanations.

2 Functions Outline

1. Create_array_datatype
2. readFromFileAllRead(datapoints, array)
3. buildTreeGlobal(array, num, &headNode, -1)
   (A) getMaxMinGlobal(array, num, colIndex, anode→max, anode→min)
   (B) getLargestDimensionGlobal(anode→max, anode→min, &colIndex);
   (C) globalSort(array, &num, colIndex, &globalNum)
      (i) do_sort(array, ∗num, colIndex);
      (ii) getallCount(∗num, colIndex, array, allCounts)
         (a) qsort(LDiv, (num_ranks)∗num_ranks, sizeof(float), compare_longfloat)
         (b) getCounts(num, colIndex, array, L, totalCount, allCounts)
         (c) checkBalance(&balanced, totalCount)
         (d) adjustL(num, colIndex, array, L, allCounts, totalCount, &balanced)
      (iii) struct data_struct ∗recv_array = AllToAllSend(array, &total_recv_counts, allCounts)
      (iv) do_sort(recv_array, ∗num, colIndex)
   (D) splitRanks()
   (E) getMaxMin(array, num, -1, anode→max, anode→min)
4. globalTreeMaster(&Gtree, localHead)
5. readFromFile(fname, targetSize, targetArray)
6. buildTree(array, num, localHead, -1)
7. getSendArray(&Gtree,0.1, targetArray, targetSize,sendArray, sendSize[i],i)
8. localSearch(localHead, sendArray[sendi], childArray, &radiCounts[radi])

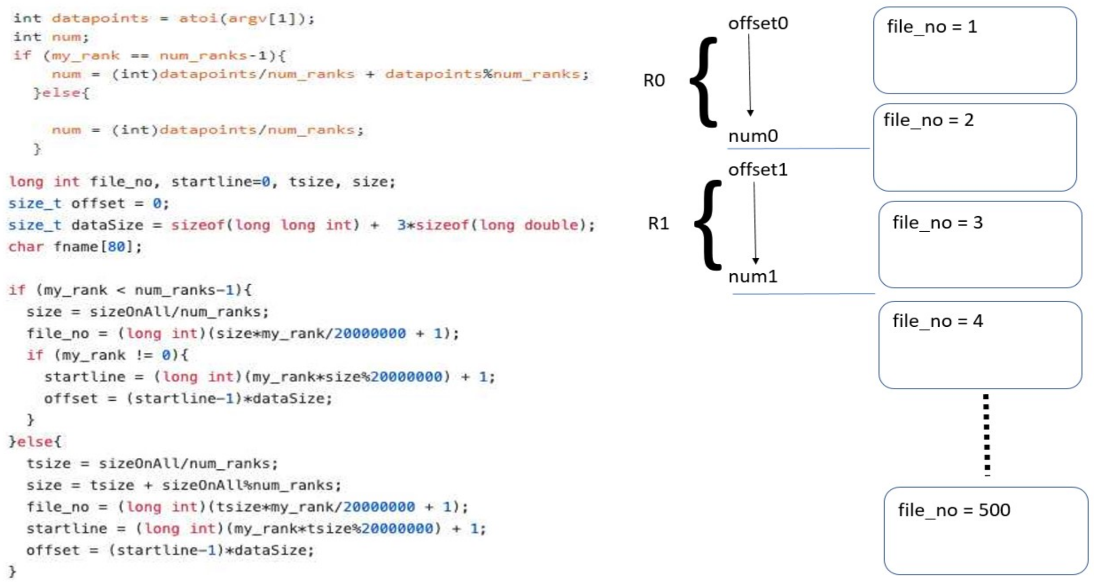
3 Create new MPI Datatypes

We created three new MPI datatypes for MPI_FLOAT, MPI_LONG_INT and for packing the struct. These new datatypes are used in passing messages from one ranks to the other.

```
void create_array_datatype(){
  MPI_Datatype data_type[2];
  int data_length[2];
  MPI_Aint displ[2], lower_bound, extent;

  MPI_Type_create_resized(MPI_LONG_INT, 0, sizeof(long int), &li_type);
  MPI_Type_commit(&li_type);

  MPI_Type_create_resized(MPI_FLOAT, 0, sizeof(float), &ld_type);
  MPI_Type_commit(&ld_type);

  // Describe the MPI_LONG_INT field in the struct
  displ[0] = 0;
  data_type[0] = li_type; //MPI_LONG_INT;
  data_length[0] = 1;

  //Describe the MPI_FLOAT in the field.
  //Obtain offset using size of MPI_LONG_INT already described
  MPI_Type_get_extent(li_type, &lower_bound, &extent);
  displ[1] = data_length[0] * extent;
  data_type[1] = ld_type; //MPI_FLOAT;
  data_length[1] = 3;

  // Define the new data_type of our struct and commit
  MPI_Type_create_struct(2, data_length, displ, data_type, &array_type);
  MPI_Type_commit(&array_type);
}
```

4 Read Points

The points are read in over a 10 GB bandwidth network. Tests have shown that there is a maximum number of 5 ranks that can read at a time. The original dataset is stored in ASCII files. In order to optimize the read, the data is converted to binary files and all ranks attempt to read a calculated portion of the files at the the same time as shown in fig. 1.



```
int datapoints = atoi(argv[1]);
int num;
if (my_rank == num_ranks-1){
    num = (int)datapoints/num_ranks + datapoints%num_ranks;
  }else{

    num = (int)datapoints/num_ranks;
  }

long int file_no, startline=0, tsize, size;
size_t offset = 0;
size_t dataSize = sizeof(long long int) +  3*sizeof(long double);
char fname[80];

if (my_rank < num_ranks-1){
  size = sizeOnAll/num_ranks;
  file_no = (long int)(size*my_rank/20000000 + 1);
  if (my_rank != 0){
    startline = (long int)(my_rank*size%20000000) + 1;
    offset = (startline-1)*dataSize;
  }
}else{
  tsize = sizeOnAll/num_ranks;
  size = tsize + sizeOnAll%num_ranks;
  file_no = (long int)(tsize*my_rank/20000000 + 1);
  startline = (long int)(my_rank*tsize%20000000) + 1;
  offset = (startline-1)*dataSize;
}
```

Figure 1: Reading all the datafiles.

The ASCII files are provided with a known format. There are 501 files with a specific filename that increments. Each file contained 20,000,000 (20 million) records. Each record or line contained an integer identifier and 3 decimal coordinates of a point.

Conversion from ASCII to binary took roughly 10 hours using a C script. The integer is stored as a long long int and the decimal numbers are stored as long doubles in the binary files. Each binary file used the same naming convention as the ASCII files.

The `readFromFileAllRead` function implements the method specified in fig. 1 to read the binary files. Based on the specified number of total data points, each rank calculated it's starting file and offset. From it's offset, each rank continued reading until it's specified number of assigned data points. To save on memory, the long double points are stored in a float type.

5 Get Local Head

The `buildTreeGlobal` function finds the local head of each rank by building a spatial bisection tree structure across all ranks. The tree structure is represented as a collection of nested boxes. The entire set of points across all ranks is enclosed inside the biggest box, ie. the global head of the tree. The global head is split into two boxes in it's largest dimension creating it's "children". Each child is then split in it's largest direction until the child is the local head for an individual rank (see fig. 2), ie. the leaves of the global tree.

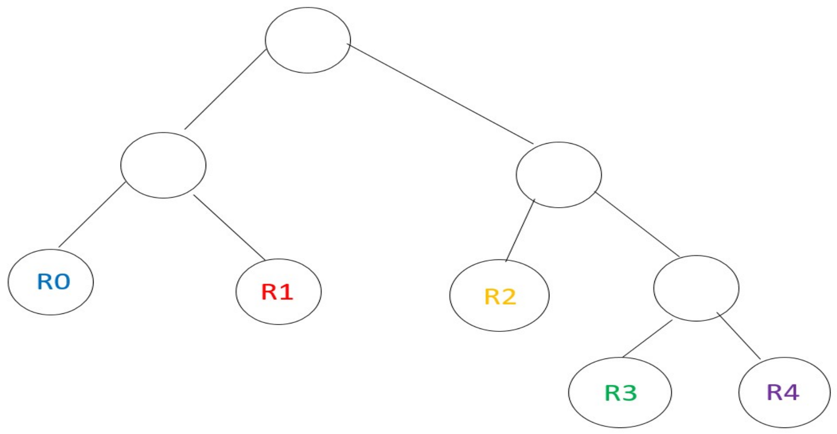


Figure 2: Conceptual Global Tree.

Each box is represented by a node (see fig. 3). The node contains the max and min values of the box, the number of elements in the box, references to the child boxes, and a pointer to the data pointed to by the leaf nodes. The max and min are representations of the dimensions of the box. The number of elements in the box are the number of ranks covered by the node. The leaves of the tree have no children and other nodes have no data.

```
struct node{
    float max[3], min[3];
    int num_below;
    struct node *left, *right;
    struct data_struct *center;
}
```

Figure 3: Local Node Struct

This process requires 4 main steps. The dimensions of the box or the max and min should be found. The largest dimension is then obtained. The points must be sorted across all nodes. Finally, the box is split in it's largest dimension.

In order to get the max/min, sort, and split the "box", several global variables are used. Each "box" accesses a global communicator and number of ranks variable. The management of communicators is a critical portion of the timing of the code. Each rank also maintains it's id in a box communicator and in the global communicator through two rank variables.

The dimensions of the box do not need to be exact representations of the points. In the global head, the max and min are literally the largest and smallest values in all dimensions. When a node is split in a particular dimension, the split dimension is the only max and min that is updated. All other dimensions are inherited from the parent. This can lead to the other dimensions being estimations of the points in the box.

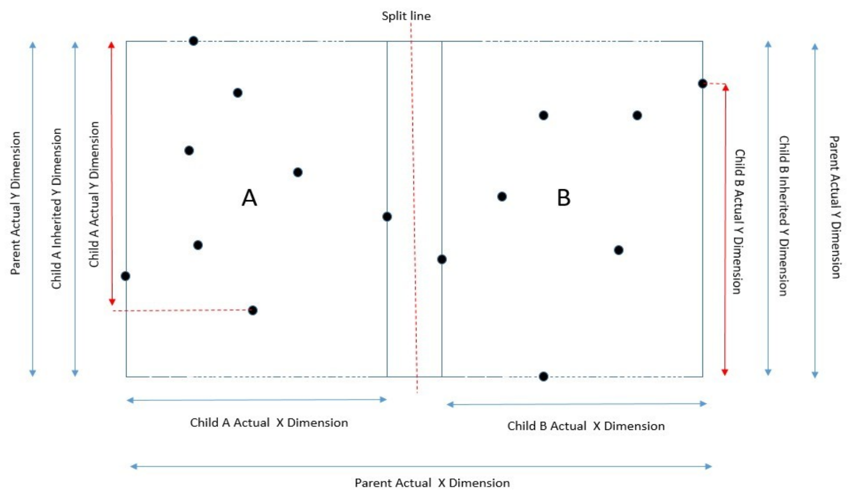

Split line
Parent Actual Y Dimension
Child A Inherited Y Dimension
Child A Actual Y Dimension
A
B
Child B Actual Y Dimension
Child B Inherited Y Dimension
Parent Actual Y Dimension
Child A Actual X Dimension
Child B Actual X Dimension
Parent Actual X Dimension

Figure 4: Split Dimension

The `getMaxMinGlobal` function is used to find the max and min across all ranks. This function accepts a direction or column index as an argument. When the column index is not 0, 1, or 2, it finds the max and min in all dimensions. Otherwise, it updates the specified dimension. The function performs sequential search on each rank. All ranks compare the dimensions from all other ranks to find the max and min across all ranks.

A `getLargestDimensionGlobal` function is used to find the largest dimension. This function use a simple loop to find the the dimension with the largest difference between the max and min.

The global sort is an in depth process described in the next section. In the `globalSort` function, the points are only sorted in the largest dimension. The sort is meant to evenly divide the points onto all the ranks represented in a single node.

The split in a K-D tree typically divides the data into 2 equally sized groups. In the `splitRanks` function, the divide is between ranks. When there are an uneven number of ranks, the "right" child always receives the remaining rank. The order of the ranks is preserved from their original identifier. When a split is completed, the communicator and global management variables are also updated.

Algorithm 1 Get Local Head

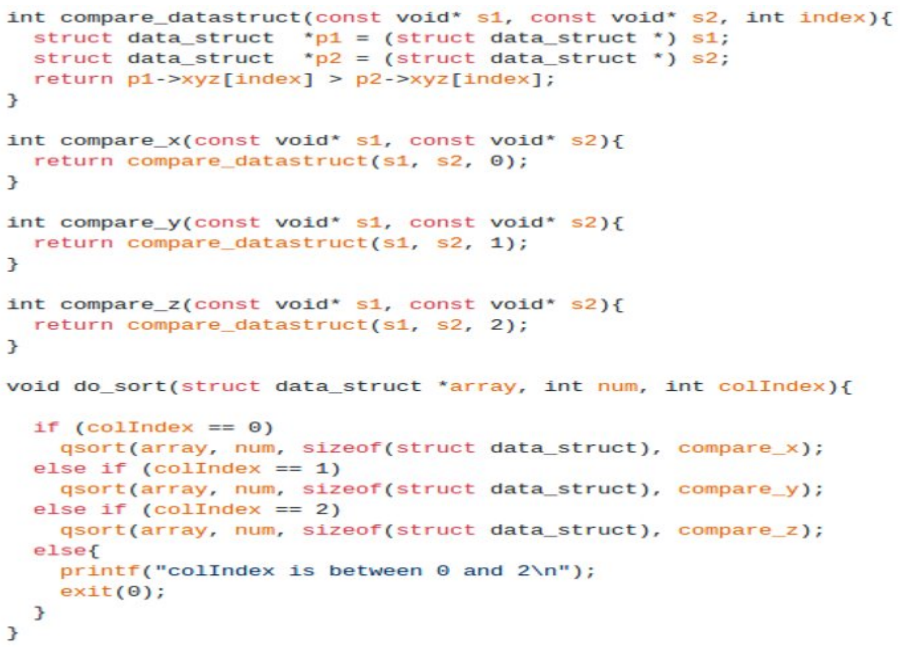
```
struct node *anode;
int datapoints = atoi(argv[1]);
struct data_struct* array = (struct data_struct *) malloc(num * sizeof(struct
data_struct));
readFromFileAllRead(datapoints, array );
if my_rank == global_num_ranks-1 then
    num = (int)datapoints/num_ranks + datapoints%num_ranks;
else
    num = (int)datapoints/num_ranks;
end if
while num_ranks > 0 do
    getMaxMinGlobal(array, num, colIndex, anode→max, anode→min);
    getLargestDimensionGlobal(anode→max, anode→min, &colIndex);
    array = globalSort(array, &num, colIndex, &globalNum);
    splitRanks();
end while
```

5.1 Global Sort

The global sort effectively moves points from 1 rank to another such that the points are ordered in a specified dimension over the ranks according to the rank's communicator identifier. In the end, the smallest value in the specified dimension on any rank is greater than the biggest value in the specified dimension on the ranks neighbor to the "left". These smallest values are identified as the lower bounds of each rank. Conceptually, the global sort finds the lower bounds such that each rank has an equal number of points.

The global sort is completed using 4 steps: local sort, get counts, send points, and local sort. The local sort is completed in a specified dimension using an intrinsic C function. The get counts uses a bucketing algorithm to find the lower bounds and the number of points on each rank. The send points uses an MPI call to transfer the number of points specified with the bucketing algorithm to each rank.

The `do_sort` function completes the local sort using the intrinsic C function `qsort`. The `qsort` function has a comparison function as a parameter. The comparison function must only have two parameters: the two elements being compared. When a local sort is completed, it requires a 3rd parameter: the sort dimension. A comparison function is defined for each dimension. The `do_sort` function accepts the 3rd parameter and chooses the appropriate comparison function.



```
int compare_datastruct(const void* s1, const void* s2, int index){
  struct data_struct  *p1 = (struct data_struct *) s1;
  struct data_struct  *p2 = (struct data_struct *) s2;
  return p1->xyz[index] > p2->xyz[index];
}

int compare_x(const void* s1, const void* s2){
  return compare_datastruct(s1, s2, 0);
}

int compare_y(const void* s1, const void* s2){
  return compare_datastruct(s1, s2, 1);
}

int compare_z(const void* s1, const void* s2){
  return compare_datastruct(s1, s2, 2);
}

void do_sort(struct data_struct *array, int num, int colIndex){

  if (colIndex == 0)
    qsort(array, num, sizeof(struct data_struct), compare_x);
  else if (colIndex == 1)
    qsort(array, num, sizeof(struct data_struct), compare_y);
  else if (colIndex == 2)
    qsort(array, num, sizeof(struct data_struct), compare_z);
  else{
    printf("colIndex is between 0 and 2\n");
    exit(0);
  }
}
```

Figure 5: Local Sort

The `getallCount` function completes the get counts and implements the bucketing algorithm. This process is defined in two primary steps: guess the lower limits and adjust the lower limits. Both steps count the number of elements that would end up on each rank and compares that number to the expected number of each rank. A collection of variables are defined to help understand the process below.

```
array:  holds all the elements on this rank
num: the size of array
MPI_LOCAL_COMM: the communicator that contains all ranks in this node
num_ranks: the number of ranks in MPI_LOCAL_COMM
**K = (int)num/num_ranks: the target number of elements on each rank
l: the local lower limits and the maximum on each rank (size = num_ranks+1)
L: the global lower limits and the global maximum (size = num_ranks+1)
```

The primary purpose of `getallCount` is to guess the lower limits. The first guess is made by using a 3 step process: select the lower limits that split the data into `num_ranks` buckets, sort those lower limits across all ranks, and select every `num_ranks`th lower limit as the global lower limit. In practice, the maximum is also required to complete certain parts of the code. The variable that holds the lower limits also holds the maximum. The Guess Buckets figure below visualizes how this guess works.

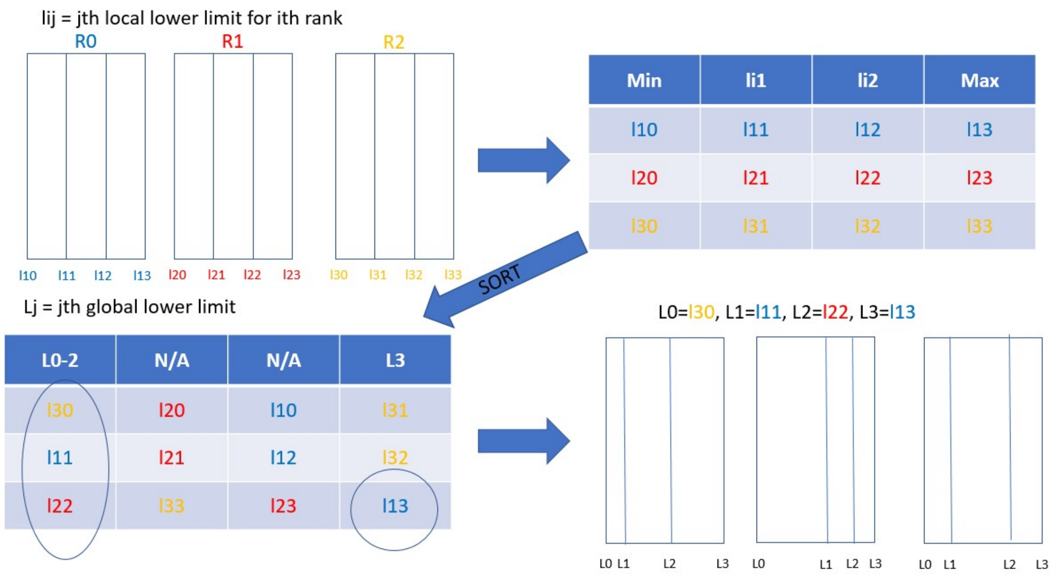


Figure 6: Guess Buckets

The global lower limits are used to count the points that would be on each rank after the data is exchanged. The `getCounts` function completes this process. Each rank counts all the points that it needs to send to all ranks (including itself). The master rank puts all those counts in a single array named `allCounts`. The rows of `allCounts` contain the counts from each rank. The columns of `allCounts` contain the counts going to each rank. The columns of `allCounts` are summed into an array named `totalCounts`.

The `checkBalance` function checks the `totalCounts` against the target number of elements on each rank, `K`. If the difference between the count and `K` on any rank is greater than 10% of `K`, the buckets are flagged as unbalanced. The `adjustL` function adjust the global lower bounds until the buckets are balanced.

The adjustments to the global lower bounds have many check and tests to ensure adjustment is a success. The basic algorithm moves 1 global lower bound at a time until the buckets are balanced. The amount it is moved is based on the distance between global lower bound to the left and the right of the current global lower bound named `myrange`. The step size starts of as 10% of `myrange`. After the difference between the bucket size and `K` increases, the percent of `myrange` used decreases by half. If there is an out of bounds error, this process starts over. The following figures 7 and 8 visualize 1 step of this process.

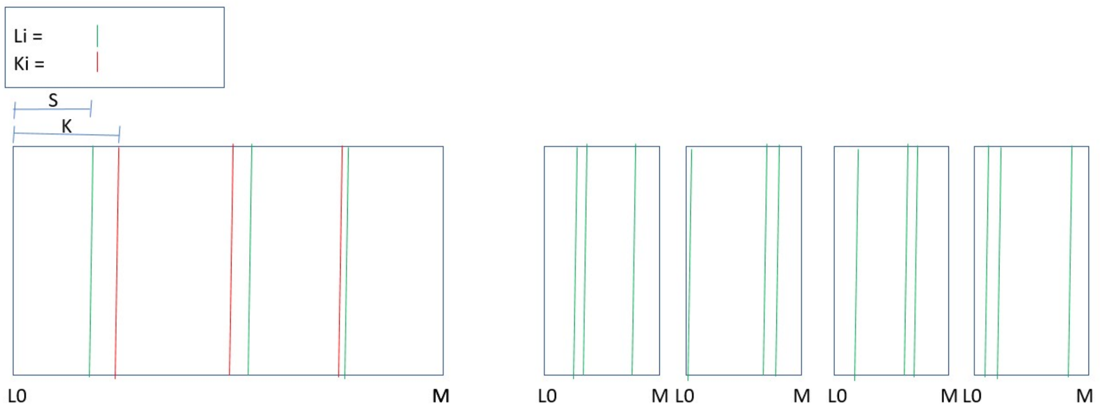


Figure 7: Adjusting buckets.

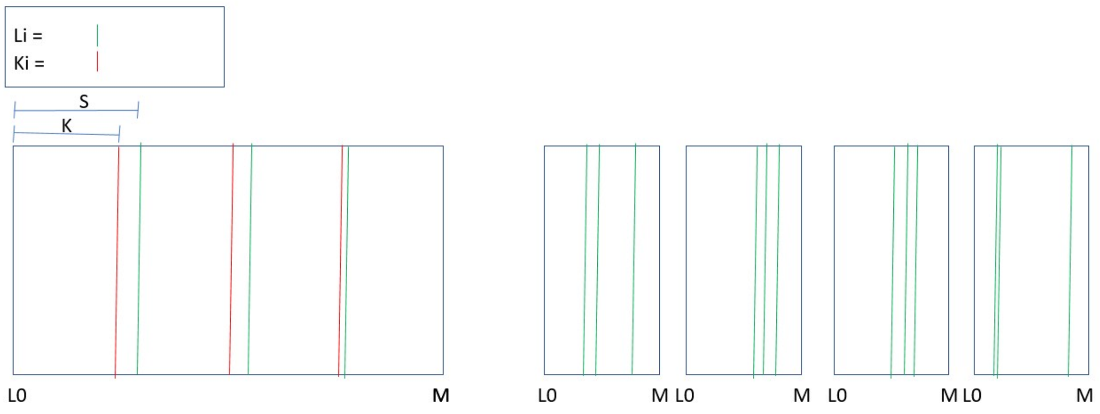


Figure 8: Adjusting buckets.

There are many other checks to the `adjustL` function. For example, the step size may not be smaller than the smallest difference between any two variables. If it is, the function attempts a few more steps to get a balanced rank or a non zero count. If 1 limit crosses over another, the are set to equal and the step size percent is set back to 10% of `myrange`.

The `AlltoAllSend` function sends points from all ranks to all ranks. It uses the `allCounts` variable to create the counts and displacements for the `MPI_Alltoall`.

6 Build Global Tree

The `globalTreeMaster` function is built only on the master rank. It uses the local heads of all ranks to build the global tree. An empty tree with global_num_ranks leaves is built. The leaves of the global tree is filled by receiving the maximum and minimum of the leaf from the other ranks in the order global_num_ranks-1, global_num_ranks-2, $\cdots$, 2, 1. Each node in the tree holds the following information.

```
struct Gnode{
float max[3], min[3];
int this_rank, num_below, assigned = -1;
struct Gnode *left, *right, *parent*;
struct data_struct *center;
}
```

While building the empty tree, the gnum_below is assigned for each node as num_below = global_num_ranks % 2. The master rank receives the maximum and minimum (from rank global_num_ranks-1) to the rightmost leaf as shown in fig. 9, and calculates the information of the node. It traverses to the parent in which case assigned equals zero (flag shown in fig. 10) indicating that the left child has not been visited. It then goes to the left child and calculates all its information. It traverses to the parent again in which case assigned equals one (okay shown in fig. 11). This process is continued until the tree is built to the global head node. The below algorithm describes the process better.

Algorithm 2 Build Global Tree

```
1: struct Gnode *Gtree;
2: currNode = buildEmptyGtree(Gtree, ranks_below, 0)
3: j = num_ranks - 1
4: while $j > 0$ do
5:     if currNode→assigned == -1 then
6:         if currNode→num_below <= 1 then
7:             Receive max and min of the leaf from rank j;
8:             currNode→this_rank = j;
9:             currNode = currNode→parent;
10:            j –;
11:        else
12:            if currNode→right→assigned == -1 then
13:                currNode = currNode→right;
14:            else if currNode→left→assigned == -1 then
15:                currNode = currNode→left;
16:            else
17:                Build the node up
18:                currNode = currNode→parent;
19:            end if
20:        end if
21:    end if
22: end while
```

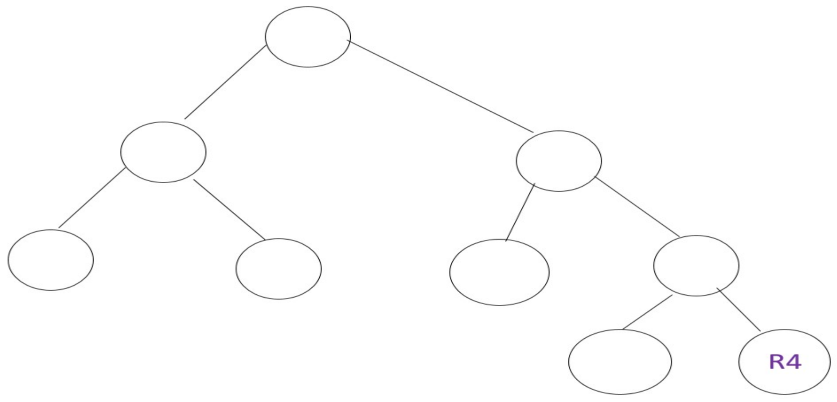


Figure 9: Build Global Tree.

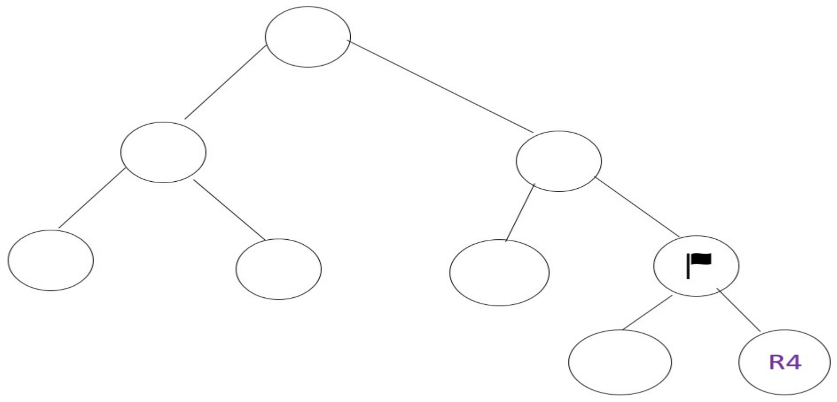


Figure 10: Build Global Tree.

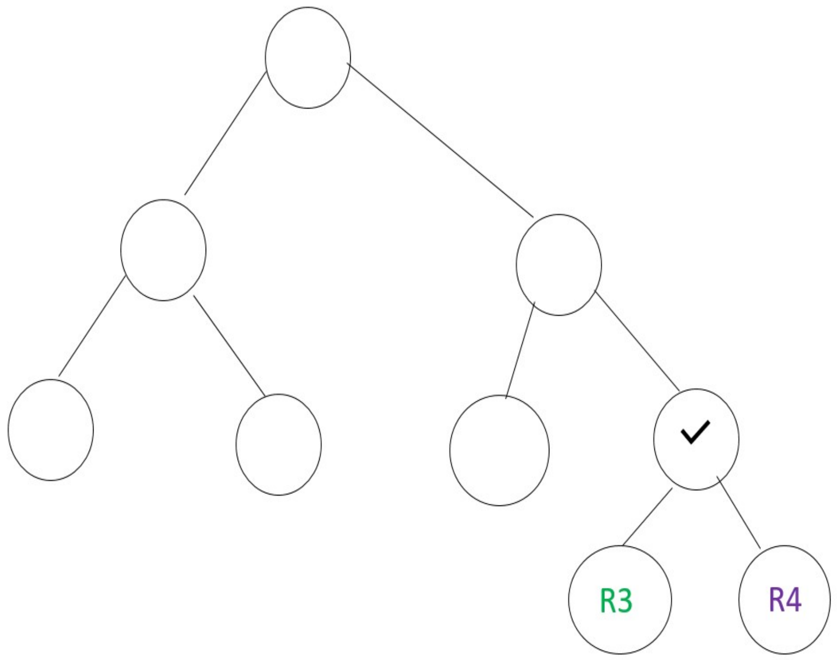


Figure 11: Build Global Tree.

7 Read Targets

In our implementation, only the master rank reads the target file named `bdatafile00501.bin` and the number of targets read is specified as a command line argument. The file is converted to binary and stored as long doubles as discussed in the `Read Points` section. It is then read using the `fread` construct as long doubles to a variable which was then converted to floats using type-casting. The code snippet below better explains the process.

```
while(!feof(fp) && i < size){
      fread(&id,sizeof(long int),1,fp);
      fread(&x,sizeof(long double),1,fp);
      fread(&y,sizeof(long double),1,fp);
      fread(&z,sizeof(long double),1,fp);

      array[i].num = id;
      array[i].xyz[0] = (float)x;
      array[i].xyz[1] = (float)y;
      array[i].xyz[2] = (float)z;
}
```

8 Get Target Size and Targets

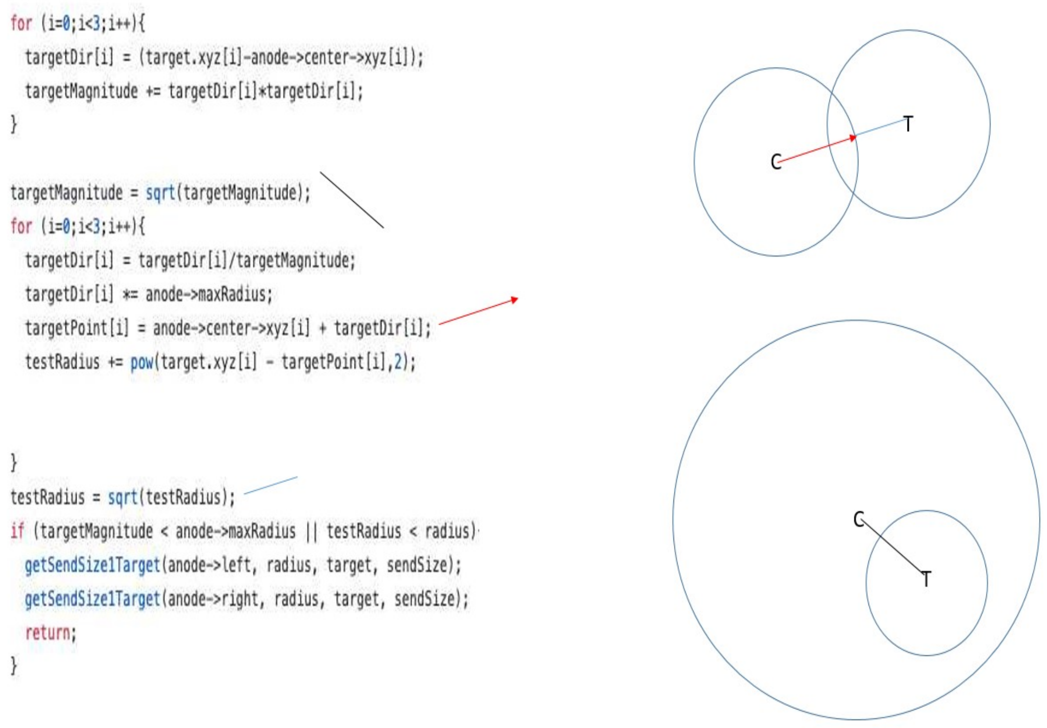
The master rank needs to send the targets that may be located on a node (within the given radii) to the node. In order to do this, the master rank has to allocate memory for the number of targets belonging to a particular node. So it traverses through the global tree to get the targets to send to each node. In particular, it only searches for the targets using the largest radius (r = 0.1) since if a target is found on a node with the largest radius, it should also be on the node using a smaller radius. The master rank allocates an array of data_struct of size `targetSize` and get the actual targets in the allocated array (sequentially for each node). The procedure discussed is better described with the pseudocode below:

```
sendArray = (struct data_struct *) malloc(targetSize* sizeof(struct data_struct));
for (i = 1; i<global_num_ranks; i++){
    getSendArray(&Gtree, 0.1, targetArray, targetSize, sendArray, &sendSize[i], i);
    MPI_Send(sendArray, sendSize[i], array_type, i, 0, MPI_COMM_WORLD);
}
```

The `getSendArray` functions is used to determine if a point in the current node (say, `anode`) is within the radius of a target (targetRadius). We define two terminologies, the targetSphere and nodeSphere, where the former is used to denote a sphere centered at the target point with the given radius and the latter is used to denote a sphere centered at `anode`→`center` with radius `anode`→`maxRadius`. There are two cases; either the target-Sphere intersect the nodeSphere or the targetSphere is contained in the nodeSphere as shown in fig. 12. For the latter case, we calculate the distance between the target and the node, and compare with `anode`→`maxRadius`. For the former case, we calculate a unit vector in the direction of the target which is then scaled by `anode`→`maxRadius`. Then, we compute a `testRadius` which is the distance between the edge (towards the target) of the nodeSphere and the targetSphere. The `testRadius` is then compared with the radius of the target.

For the other ranks, they receive the sendArray as follows:

```
    MPI_Probe(0, 0, MPI_COMM_WORLD, &stat);
    MPI_Get_count(&stat,array_type,&mySendSize);
    sendArray = (struct data_struct *) malloc(mySendSize * sizeof(struct data_struct));
    MPI_Recv(sendArray, mySendSize, array_type, 0, 0, MPI_COMM_WORLD, &mystat);
```

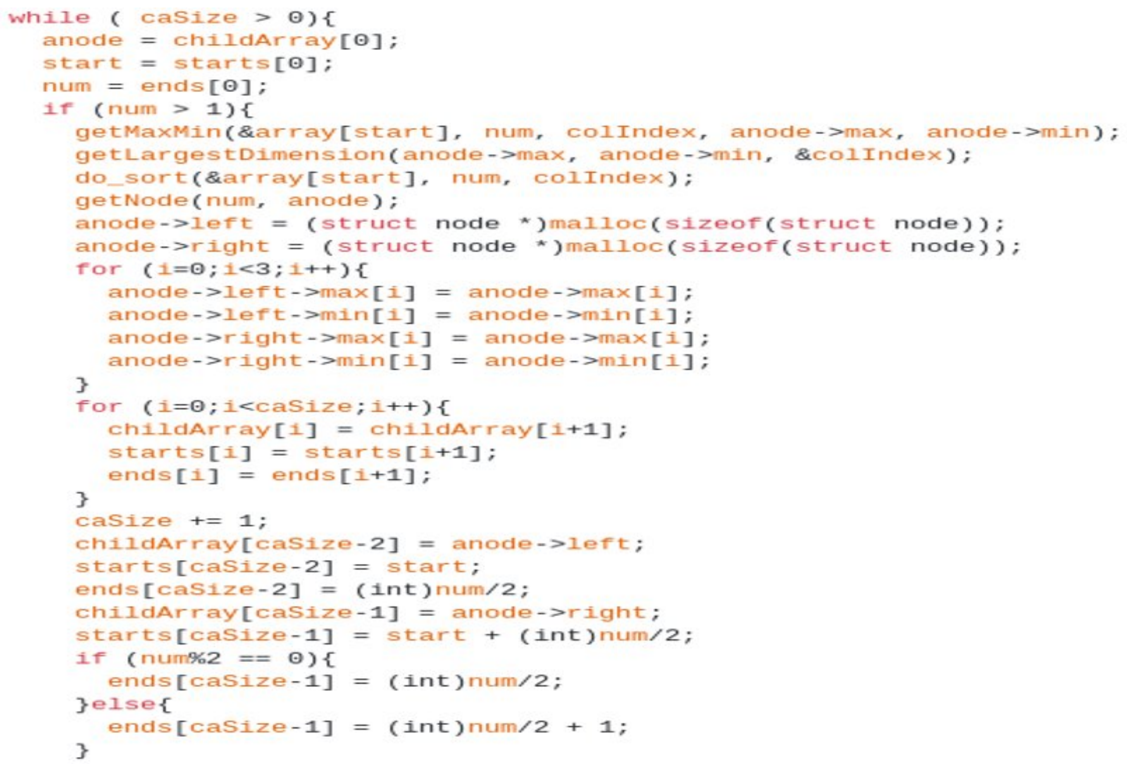


```
for (i=0;i<3;i++){
  targetDir[i] = (target.xyz[i]-anode->center->xyz[i]);
  targetMagnitude += targetDir[i]*targetDir[i];
}

targetMagnitude = sqrt(targetMagnitude);
for (i=0;i<3;i++){
  targetDir[i] = targetDir[i]/targetMagnitude;
  targetDir[i] *= anode->maxRadius;
  targetPoint[i] = anode->center->xyz[i] + targetDir[i];
  testRadius += pow(target.xyz[i] - targetPoint[i],2);


}
testRadius = sqrt(testRadius);
if (targetMagnitude < anode->maxRadius || testRadius < radius)
  getSendSize1Target(anode->left, radius, target, sendSize);
  getSendSize1Target(anode->right, radius, target, sendSize);
  return;
}
```

Figure 12: Target size .

9 Build Local Tree

With the local head on each node (which are the leaves of the global tree on master rank), all ranks performs a serial ORB of the datapoints in the node (called local tree) as discussed in `Get Local Head` section.

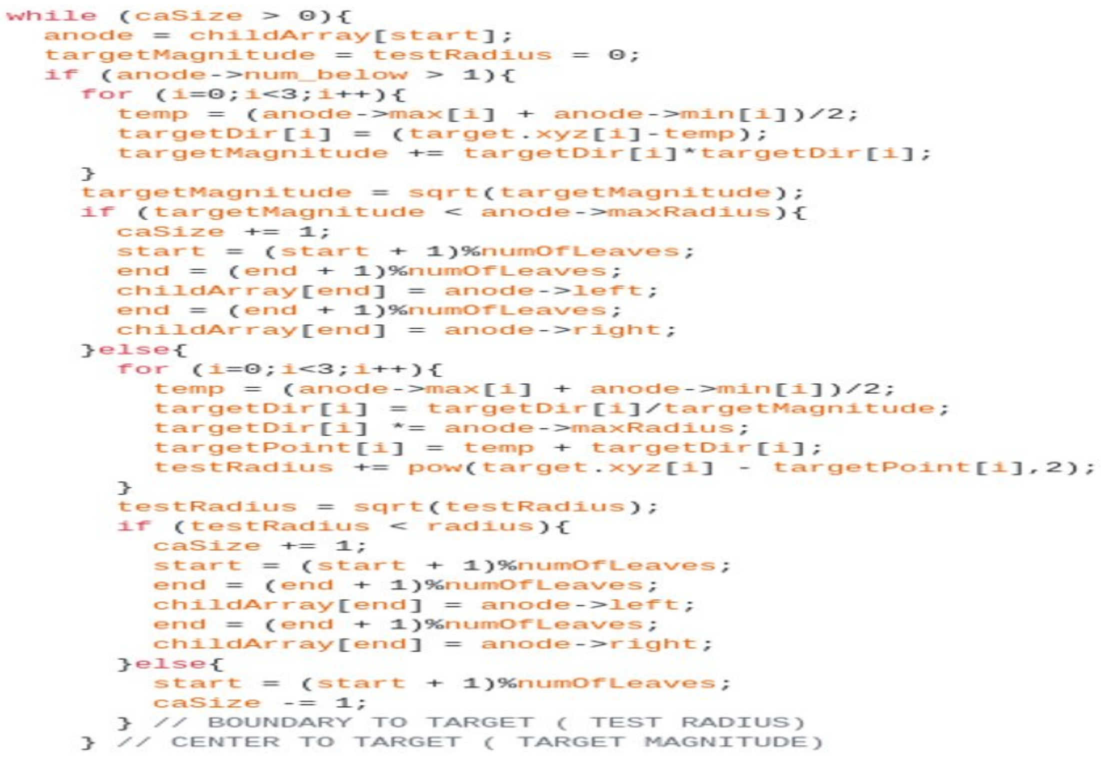


```
while ( caSize > 0){
  anode = childArray[0];
  start = starts[0];
  num = ends[0];
  if (num > 1){
    getMaxMin(&array[start], num, colIndex, anode->max, anode->min);
    getLargestDimension(anode->max, anode->min, &colIndex);
    do_sort(&array[start], num, colIndex);
    getNode(num, anode);
    anode->left = (struct node *)malloc(sizeof(struct node));
    anode->right = (struct node *)malloc(sizeof(struct node));
    for (i=0;i<3;i++){
      anode->left->max[i] = anode->max[i];
      anode->left->min[i] = anode->min[i];
      anode->right->max[i] = anode->max[i];
      anode->right->min[i] = anode->min[i];
    }
    for (i=0;i<caSize;i++){
      childArray[i] = childArray[i+1];
      starts[i] = starts[i+1];
      ends[i] = ends[i+1];
    }
    caSize += 1;
    childArray[caSize-2] = anode->left;
    starts[caSize-2] = start;
    ends[caSize-2] = (int)num/2;
    childArray[caSize-1] = anode->right;
    starts[caSize-1] = start + (int)num/2;
    if (num%2 == 0){
      ends[caSize-1] = (int)num/2;
    }else{
      ends[caSize-1] = (int)num/2 + 1;
    }
```

Figure 13: Local Tree Build

10 Local Count

Each rank uses it's local tree to find the number of datapoints within each targetSPhere (for each target and radius).
The `localsearch` function uses the same logic as the **`getSendSize`** and **`getSendArray`** functions except that it compares the leaves of the local tree with the targetSphere. For efficiency and simplicity, we traverse the local tree with the largest radius $r = 0.1$, and only compare the three given radii with the **`testRadius`** at the leaves of the tree. The number of datapoints found for each target is then sent to the master rank.



```
while (caSize > 0){
  anode = childArray[start];
  targetMagnitude = testRadius = 0;
  if (anode->num_below > 1){
    for (i=0;i<3;i++){
      temp = (anode->max[i] + anode->min[i])/2;
      targetDir[i] = (target.xyz[i]-temp);
      targetMagnitude += targetDir[i]*targetDir[i];
    }
    targetMagnitude = sqrt(targetMagnitude);
    if (targetMagnitude < anode->maxRadius){
      caSize += 1;
      start = (start + 1)%numOfLeaves;
      end = (end + 1)%numOfLeaves;
      childArray[end] = anode->left;
      end = (end + 1)%numOfLeaves;
      childArray[end] = anode->right;
    }else{
      for (i=0;i<3;i++){
        temp = (anode->max[i] + anode->min[i])/2;
        targetDir[i] = targetDir[i]/targetMagnitude;
        targetDir[i] *= anode->maxRadius;
        targetPoint[i] = temp + targetDir[i];
        testRadius += pow(target.xyz[i] - targetPoint[i],2);
      }
      testRadius = sqrt(testRadius);
      if (testRadius < radius){
        caSize += 1;
        start = (start + 1)%numOfLeaves;
        end = (end + 1)%numOfLeaves;
        childArray[end] = anode->left;
        end = (end + 1)%numOfLeaves;
        childArray[end] = anode->right;
      }else{
        start = (start + 1)%numOfLeaves;
        caSize -= 1;
      } // BOUNDARY TO TARGET ( TEST RADIUS)
    } // CENTER TO TARGET ( TARGET MAGNITUDE)
```

Figure 14: Local Count

11 Global Count

The master rank calculates the total number of datapoints within each targetSphere by adding up the number of datapoints received from the other ranks.

```
if {my_global_num_ranks == 0}{
    i = 0;
    while (i<nonZeroRanks){
          MPI_Probe(MPI_ANY_SOURCE, 123, MPI_COMM_WORLD, &stat);
          d = stat.MPI_SOURCE;
          radiCounts = (long int *)malloc(tsendSize[d]*sizeof(long int));
          MPI_Recv(radiCounts,tsendSize[d] , li_type, d, 123, MPI_COMM_WORLD, &mystat);
          i++;
          for (radi=0;radi<tsendSize[d];radi+=4){
              k = (int)(radiCounts[radi]-targetArray[0].num)*3;
              for (j=1; j<=3;j++)
                   allRadiCounts[k+j-1] += radiCounts[radi+j];
          }
          free(radiCounts);
    }
 }else{
   mySendSize *=4;
   if (mySendSize > 0)
        MPI_Send(radiCounts, mySendSize, li_type, 0, 123, MPI_COMM_WORLD);
}
```

12 System Configuration

We used COMS Babbage Cluster that has 0-19 node for Intel processor and 0-29 for Amd processor. The model for Intel processor is Intel(R) Xeon(R) CPU E52620 v3 @ 2.40GHz. The python script, ”runTests.py”, is used to run the different test by varying number of ranks, datapoints and target size. The python script creates qsub file for each run with particular target size, rank and datapoint. Our whole project code is written in C language including all the MPI functions. We have ”headerfunct.h” function that includes the most of the variable and function declarations. The makefile contains instructions to compile and link all the source code files written in C program. Further, we have python code ”killprocess.py” that kill the running process for all the used node for particular process ID.

13 Results and discussion

We tested our algorithm by varying the number of datapoints from 1000000 to 10000000 and number of nodes from 16, 32, 64, and 128. We computed the readtime getLocalHead time, buildGlobalTree time readTargets time, getSendSize time, buildLocalTree time, sendSendSize time, assignTargets time, localCount time and total time required using the time stamp command in each section of the program. Figure 15 shows the total run time required for 100000 and 1000000 datapoints with 20000 targets for 16,32,64 and 128 ranks. We can observed the reduction of total run time as the number of ranks increases for 1000000 data points. However, the total run time for 100000 for data point is not decreasing as the number of rank increases. So, we need to increase the data points to get good results for higher number of ranks. Table 1 summarize all these timing results.

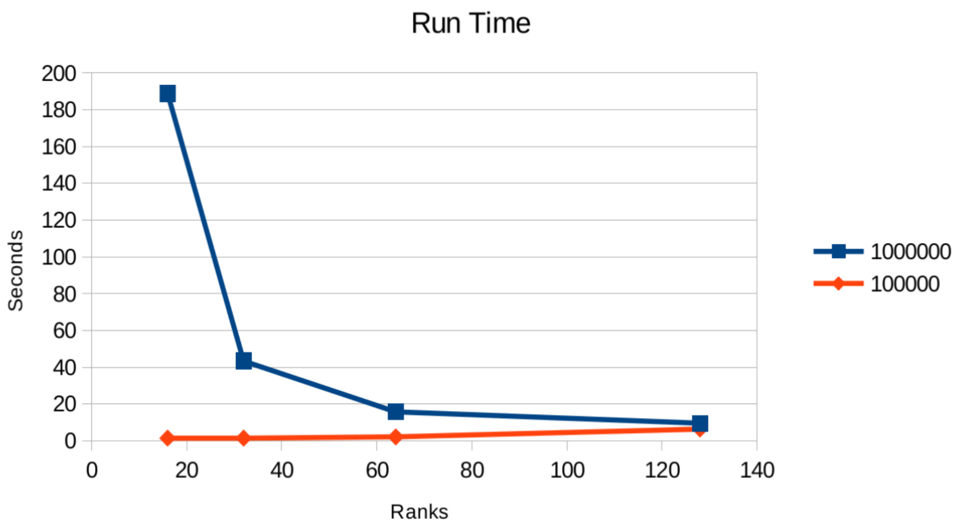


Figure 15: run time versus number of ranks for 100000 and 1000000 datapoint for 20000 data.

Table 1: Time required for 1000000 with different nodes.

Data	Nodes	read	LocalH	GloTr	Targ	LocTr	asignTarg	localC	gloC	TotT
1000000	16	0.65	0.43	0	0.02	18.37	0.01	0.00	0.02	188.82
1000000	32	0.9	0.43	0.00	0.02	5.66	0.01	0.00	0.04	43.32
1000000	64	0.5	0.61	0.00	0.01	1.42	0.01	0.00	0.03	15.64
1000000	128	0.48	1.24	0.00	0.01	0.01	0.01	0.00	0.02	9.44

Figure 16 illustrates the total run time required for different datapoints with 20000 targets for 128 ranks. We can observed the linear increments of total run time as the number of datapoints increases from 1000000 to 10000000.

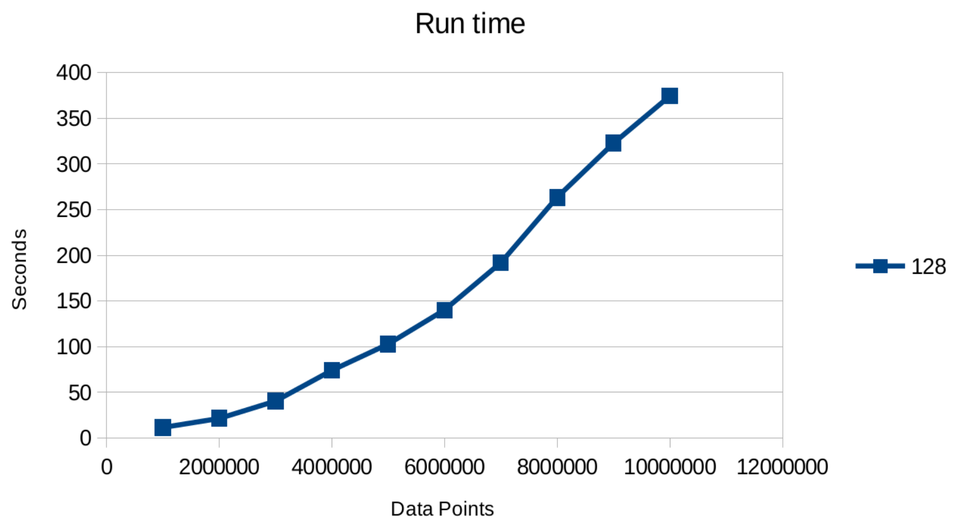


Figure 16: run time versus number of data points for 128 ranks.

14 Validation

We decided to use different validation test method to verify our output. These are the method we use for validation purpose.

- Printed out all local tree and global tree nodes.
- Matched the global tree leaves and w.r.t. local tree heads.
- For all radii, we tested file 001 and 501.
- Create the "verify. c" function for comparing all the point against all the targets.

15 Challenges

This ORB search is a really challenging project for us for this semester. We learned many MPI functions, handling large data sets, memory management, working on team projects. These are some challenges we experienced during this project implementation.

- Print statements.
- Splitting / Grouping MPI COMM WORLD.

- Cluster usage overload.
- Memory issue for large data sets and rank.
- Recursive call.
- Malloc/ Calloc.
- Global tree leaves match with local head on each rank.
- Get size of target matching, size of target array.

1. Master rank computes the total counst for each target recieved and prints the final results.

16 Conclusion

The results represent a successful implementation of an orthogonal recursive bisection (ORB) of the dataset along N-dimensions with serial and parallel sort. The algorithms and scripts are implemented for a parallel spatial binary tree that searches millions/ billion points in order to find the number of points within 3 search radii of 20 million targets. The challenges experienced are typical of a large algorithm coding project.