

Parallel Searches



Team Parallel Professional

- Group member:
 - Gabriel Toban
 - Khem Poudel
 - Toheeb Biala

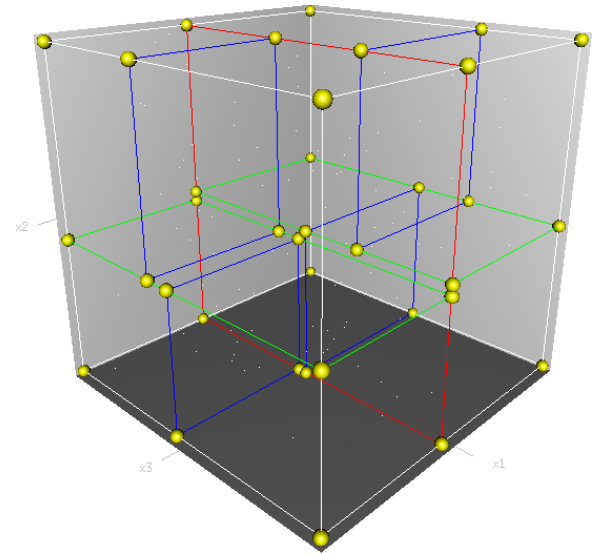


Outline

- Logic
- Code
- Results
- Challenges

Introduction

- k-d tree is a data structure used for organizing some number of points in a space with k dimensions.
- Parallel k-d tree construction requires sorting data across ranks using MPI
- Goals:
 1. Build k-d tree with 10 billion points
 2. Search k-d tree for 20 million targets
 3. Search using 3 radii



Source:Wiki

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])

Data Structures

```
struct data_struct{
    long int num;
    float xyz[3];
};
struct node{
    float max[3], min[3], maxRadius;
    struct node *left, *right;
    int num_below;
    struct data_struct *center;
};
struct Gnode{
    float max[3], min[3], maxRadius;
    struct Gnode *left, *right, *parent;
    int this_rank, num_below, assigned;
    struct data_struct *center;
};
```

New MPI data type

```
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);

    displ[0] = 0;
    data_type[0] = li_type;
    data_length[0] = 1;

    MPI_Type_get_extent(li_type, &lower_bound, &extent);
    displ[1] = data_length[0] * extent;
    data_type[1] = ld_type;
    data_length[1] = 3;

    MPI_Type_create_struct(2, data_length, displ, data_type, &array_type);
    MPI_Type_commit(&array_type);
}
```

Create Comms

```
1: MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);
2: MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
3: j = MPI_Comm_dup(MPI_COMM_WORLD, &dup_comm_world);
4: MPI_Comm_group( dup_comm_world, &world_group );
5: MPI_Comm_create( dup_comm_world, world_group, &MPI_LOCAL_COMM );
6: if num_ranks > 1 then
7:   MPI_Barrier(MPI_LOCAL_COMM)
8:   if my_rank < num_ranks/2 then
9:     cflag = 0
10:  else
11:    cflag = 1
12:  end if
13: end if
14: MPI_Comm_split(MPI_LOCAL_COMM , cflag, my_rank, &MPI_TEMP_COMM );
15: MPI_Comm_free(&MPI_LOCAL_COMM);
16: MPI_Comm_dup(MPI_TEMP_COMM, &MPI_LOCAL_COMM);
17: MPI_Comm_free(&MPI_TEMP_COMM);
18: MPI_Comm_size(MPI_LOCAL_COMM, &num_ranks);
19: MPI_Comm_rank(MPI_LOCAL_COMM, &my_rank);
```

Tree Traversal: 4 Methods

A. Recursive

- Relic
- Most Memory Usage

B. Sequential Flags

- Used for Global Tree Build Rank 0
- Traverses Tree from Leaf to Head
- Requires a flag variable in struct
- Extra Memory Usage

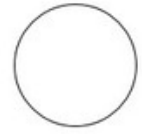
C. Sequential Array: Adjust Index

- Used for targetSize and targetArray
- Elements are moved around child array
- Slower than Round Robin by 10^2

D. Sequential Array: Round Robin

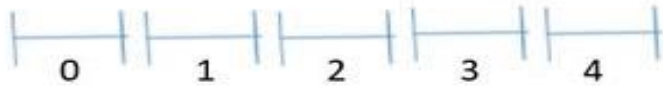
- Used for local search
- Element indexes managed in child array

Sequential Flag Method

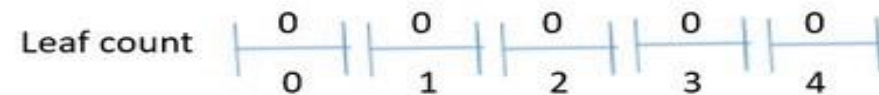
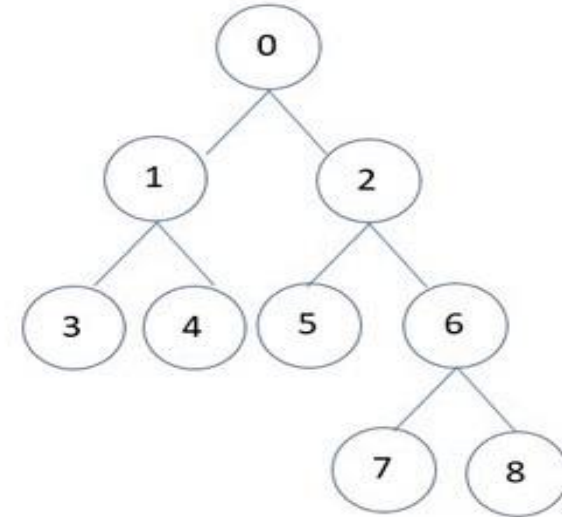
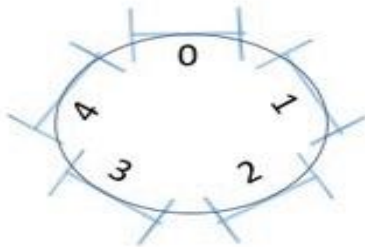


Sequential Array Methods

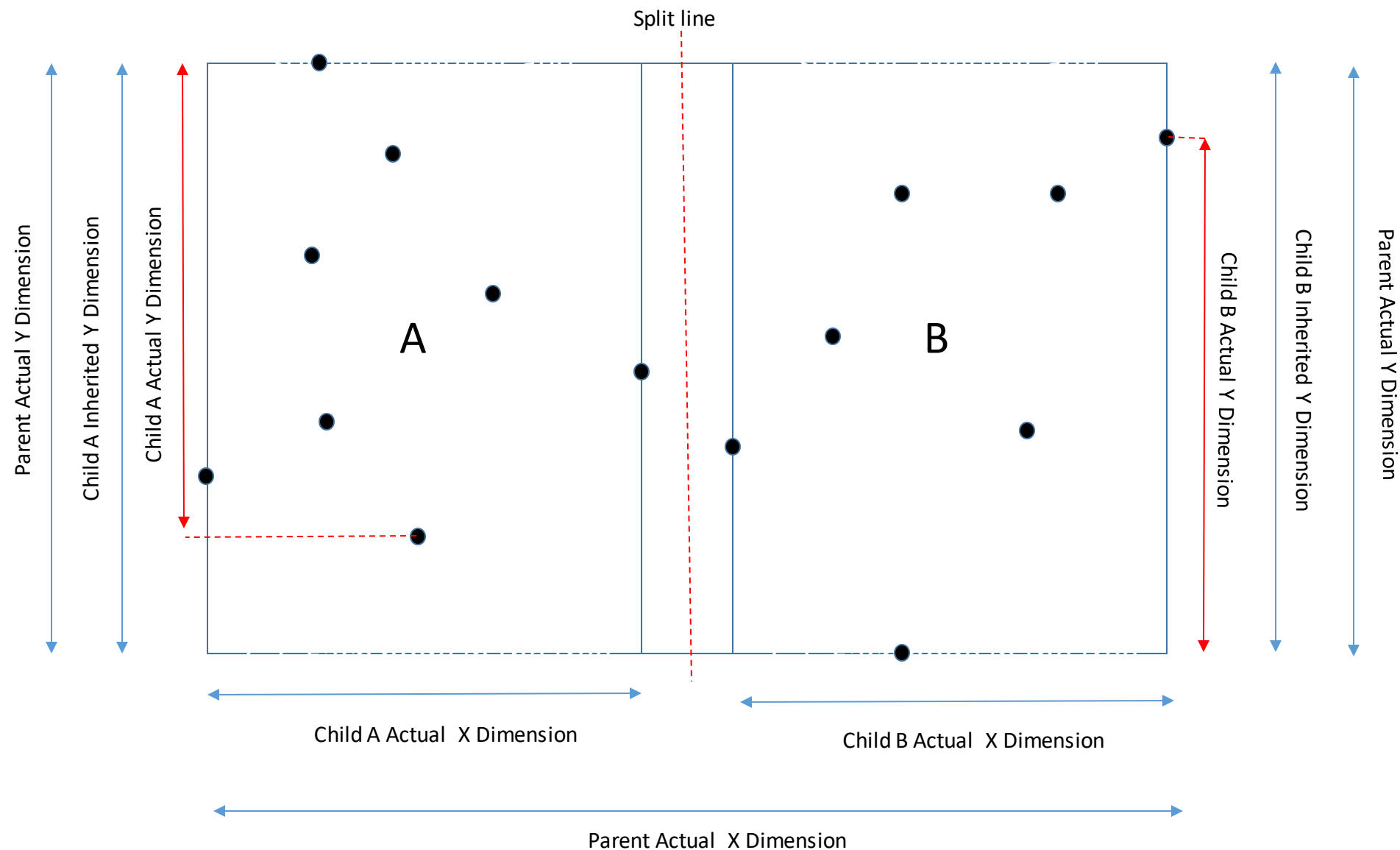
Adjust Position



Round Robin



Node Split Logic

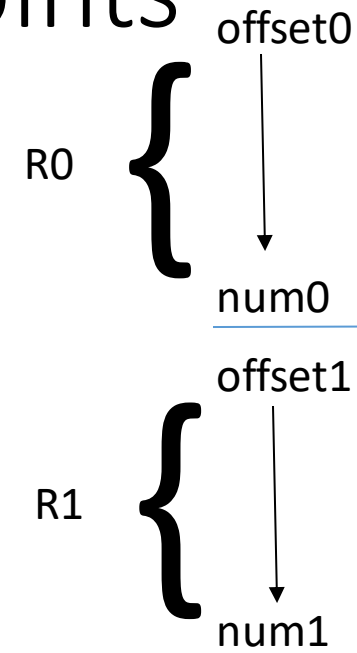


Read data points

```
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;
}
```



file_no = 1

file_no = 2

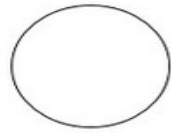
file_no = 3

file_no = 4

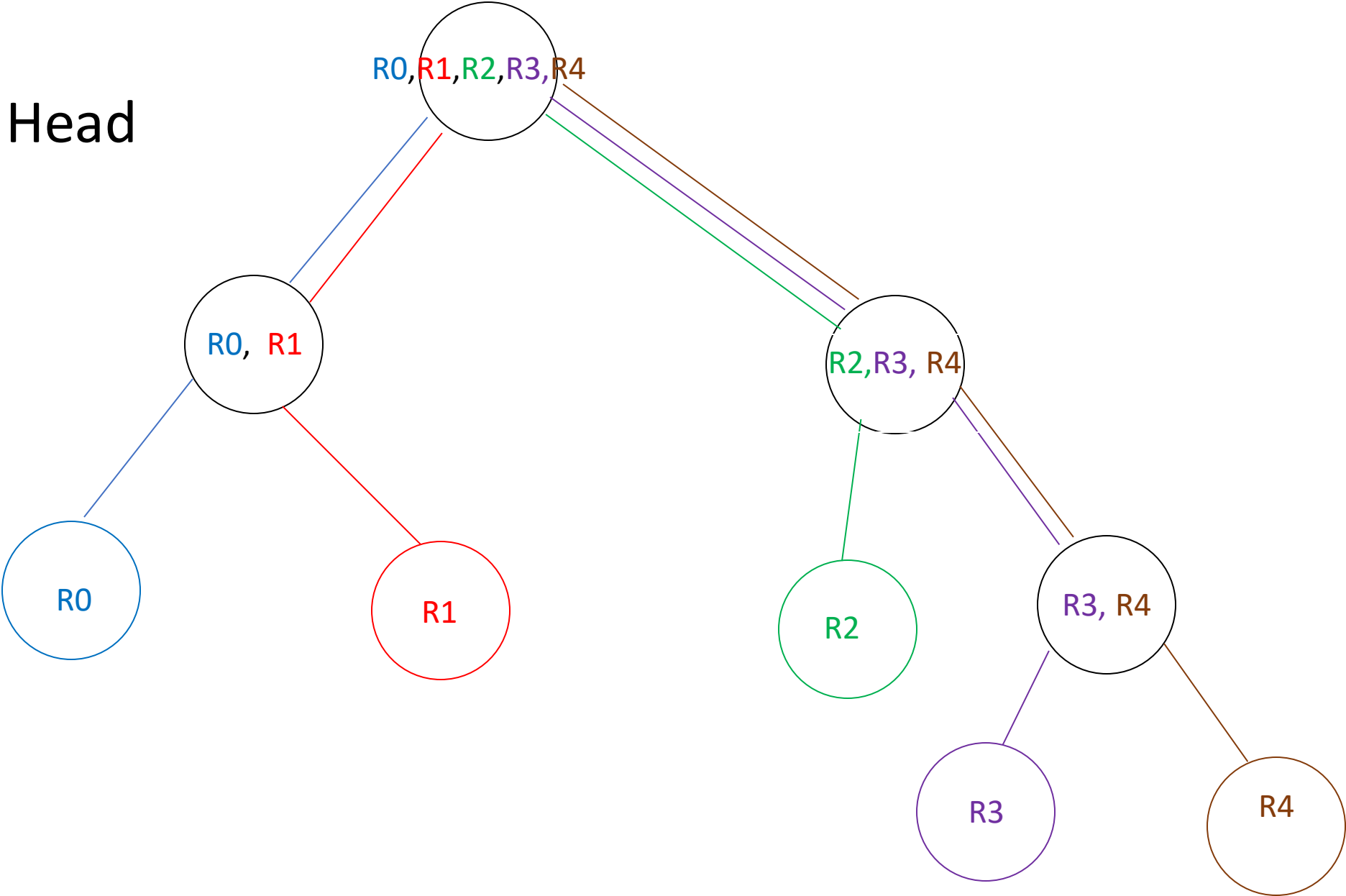
...

file_no = 500

Get Local Head



Get Local Head



Get Local Head

Algorithm 1 Get Local Head

```
1: struct node *anode;
2: int datapoints = atoi(argv[1]);
3: struct data_struct* array = (struct data_struct *) malloc(num * sizeof(struct
   data_struct));
4: readFromFileAllRead(datapoints, array );
5: if my_rank == global_num_ranks-1 then
6:   num = (int)datapoints/num_ranks + datapoints%num_ranks;
7: else
8:   num = (int)datapoints/num_ranks;
9: end if
10: while num_ranks > 0 do
11:   getMaxMinGlobal(array, num, colIndex, anode→max, anode→min);
12:   getLargestDimensionGlobal(anode→max, anode→min, &colIndex);
13:   array = globalSort(array, &num, colIndex, &globalNum);
14:   splitRanks();
15: end while
```

GetMaxMinGlobal

```
void getMaxMinGlobal(void* varray, int size, int colIndex, float *arrayMax, float *arrayMin){
    struct data_struct* array = (struct data_struct *)varray;
    float *allMax = (float *) malloc(3 * num_ranks * sizeof(float));
    float *allMin = (float *) malloc(3 * num_ranks * sizeof(float));

    int i,j,k;
    if (colIndex < 0){
        for (i=0;i<3;i++){
            arrayMax[i] = array[0].xyz[i];
            arrayMin[i] = array[0].xyz[i];
        }

        for (i=1;i<size;i++){
            for (j=0;j<3;j++){
                if (arrayMax[j] < array[i].xyz[j])
                    arrayMax[j] = array[i].xyz[j];
                else if (arrayMin[j] > array[i].xyz[j])
                    arrayMin[j] = array[i].xyz[j];
            }
        }
    }else{
        arrayMax[colIndex] = array[0].xyz[colIndex];
        arrayMin[colIndex] = array[0].xyz[colIndex];

        for (i=1;i<size;i++){

            if (arrayMax[colIndex] < array[i].xyz[colIndex])
                arrayMax[colIndex] = array[i].xyz[colIndex];
            else if (arrayMin[colIndex] > array[i].xyz[colIndex])
                arrayMin[colIndex] = array[i].xyz[colIndex];

        }
    }

    MPI_Allgather(arrayMax, 3, ld_type, allMax, 3,ld_type, MPI_LOCAL_COMM);
    MPI_Allgather(arrayMin, 3, ld_type, allMin, 3,ld_type, MPI_LOCAL_COMM);
}
```

Get Largest Dimension

```
void getLargestDimensionGlobal(float *arrayMax, float *arrayMin, int *colIndex){
    float range = arrayMax[0] - arrayMin[0];
    *colIndex = 0;
    if (range < (arrayMax[1] - arrayMin[1])){
        *colIndex = 1;
        range = arrayMax[1] - arrayMin[1];
    }
    if (range < (arrayMax[2] - arrayMin[2])){
        *colIndex = 2;
        range = arrayMax[2] - arrayMin[2];
    }
}
```

Global Sort

- localsort
- getBuckets
- adjustBuckets
- SendItems
- localsort

localsort

```
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);
}

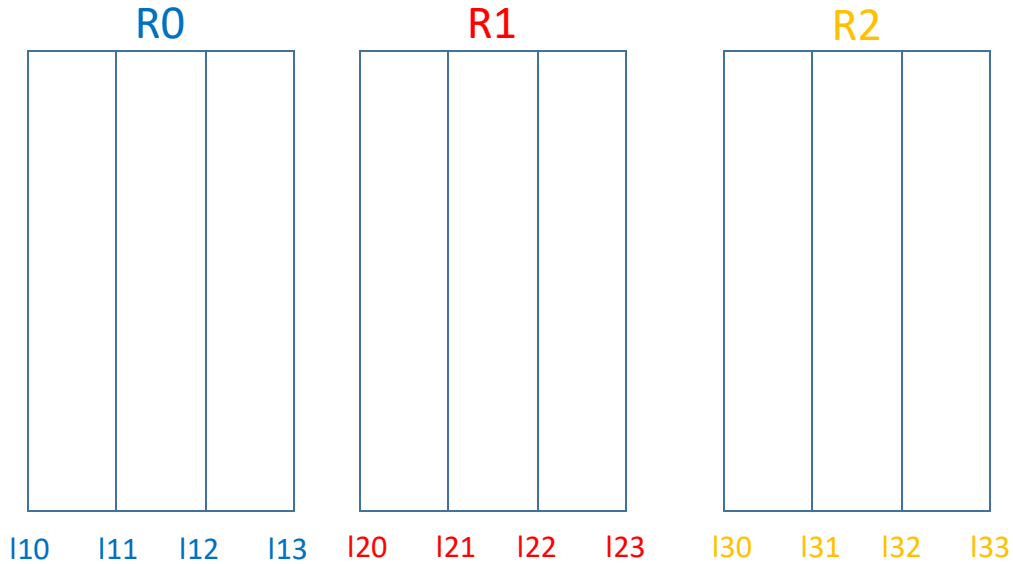
int compare_y(const void* s1, const void* s2){
    return compare_datastruct(s1, s2, 1);
}

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);
    }
}
```

GetBuckets

l_{ij} = j th local lower limit for i th rank

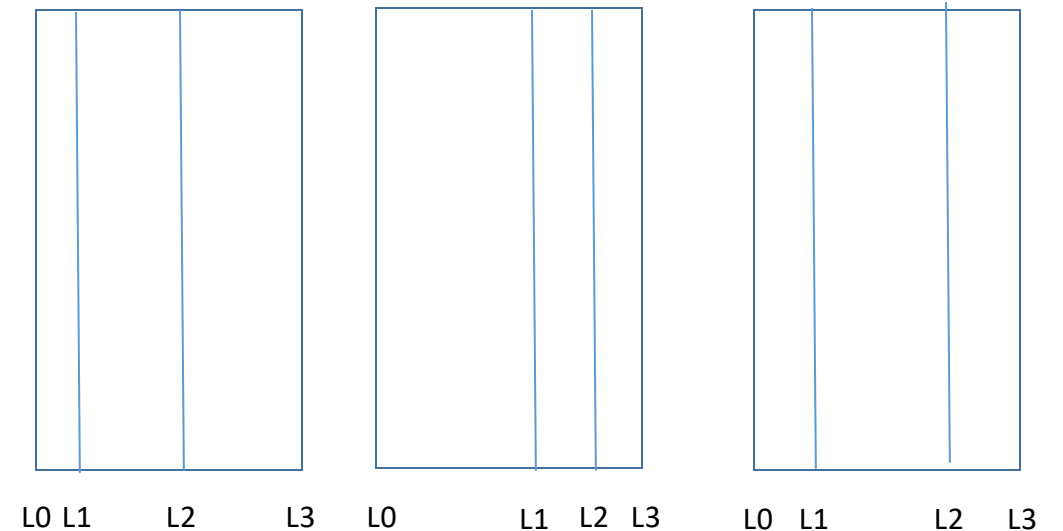


L_j = j th global lower limit

L0-2	N/A	N/A	L3
l_{30}	l_{20}	l_{10}	l_{31}
l_{11}	l_{21}	l_{12}	l_{32}
l_{22}	l_{33}	l_{23}	l_{13}

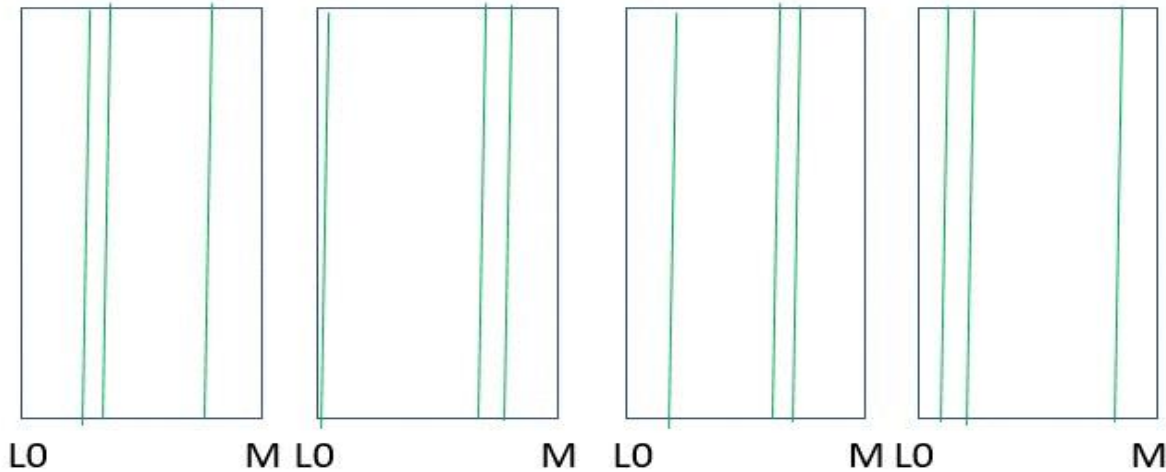
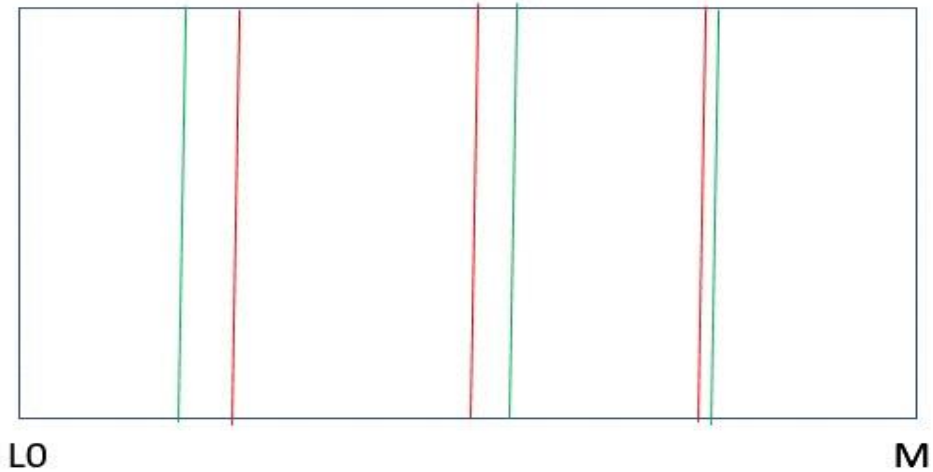
SORT

$L0=l_{30}$, $L1=l_{11}$, $L2=l_{22}$, $L3=l_{13}$

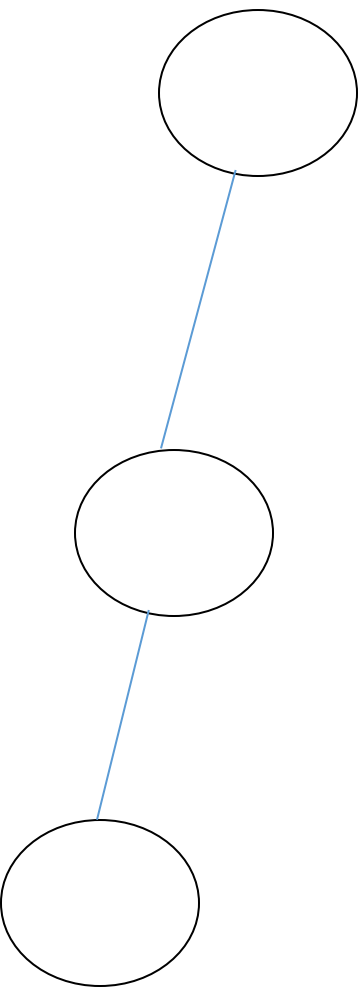


AdjustBuckets

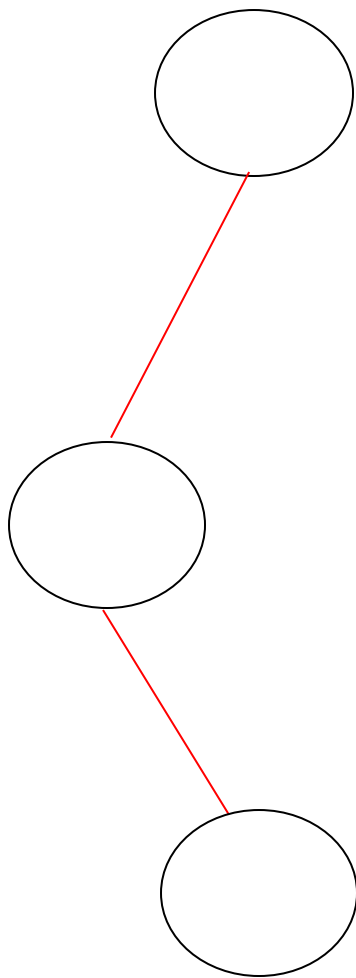
Li = 
Ki = 



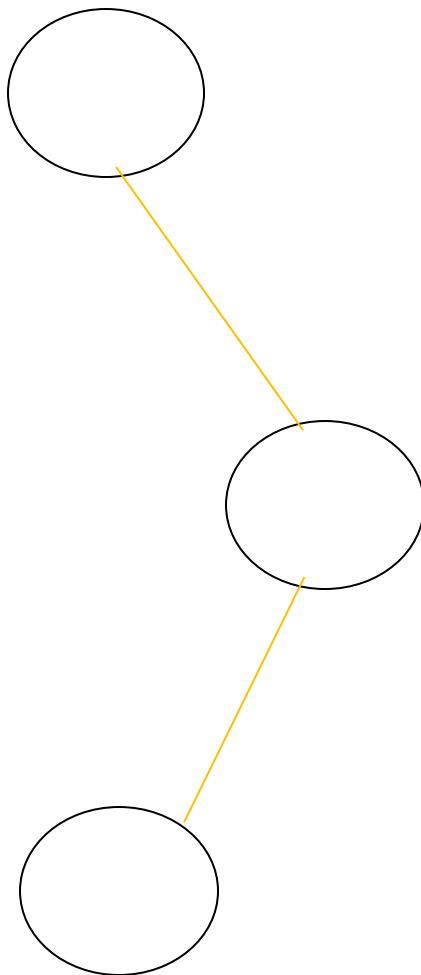
R0



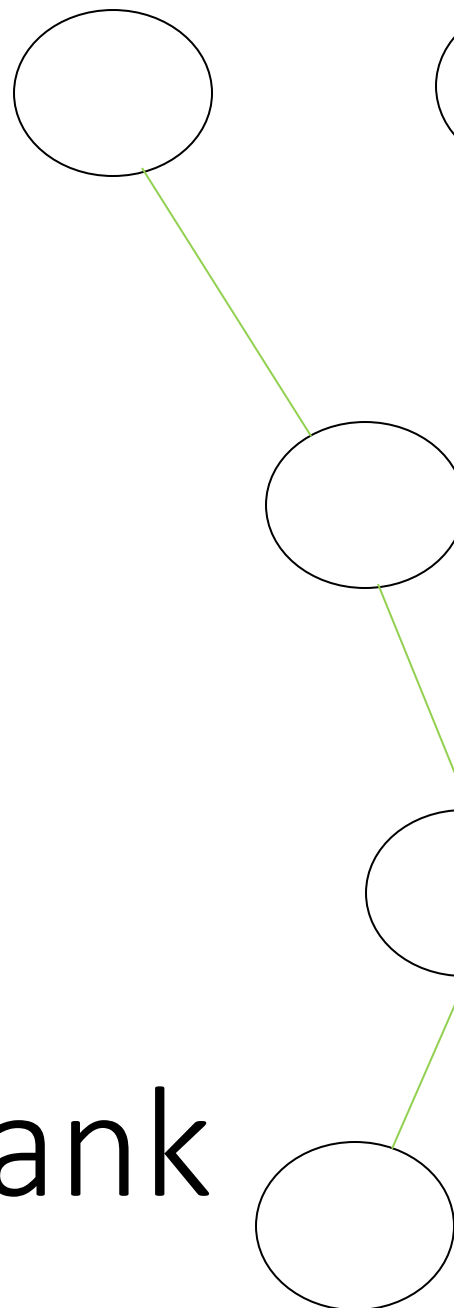
R1



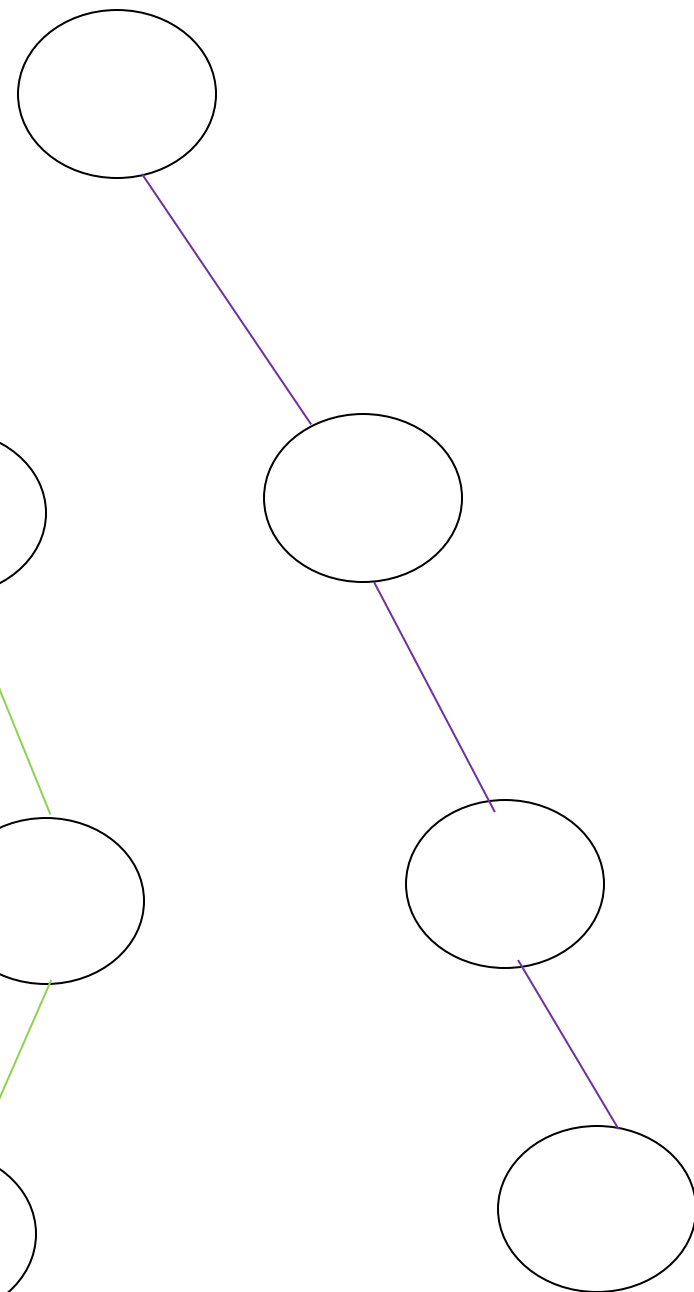
R2



R3

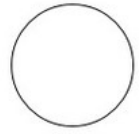


R4

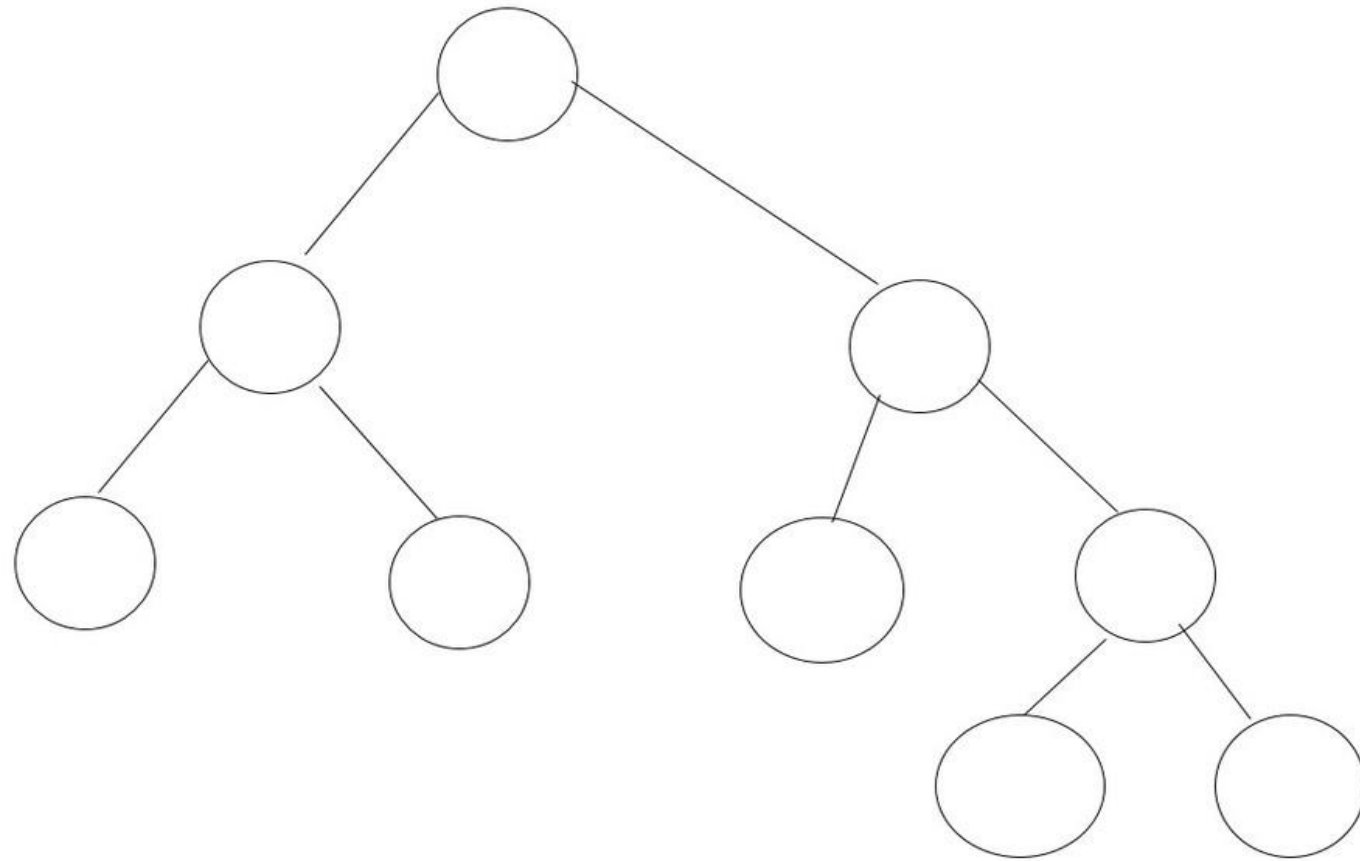


Global Tree on Each Rank

Build Global Tree Rank 0 Down



Build Global Tree up



Build Global Tree (Sequential Flag)

Algorithm 3 Build Global Tree

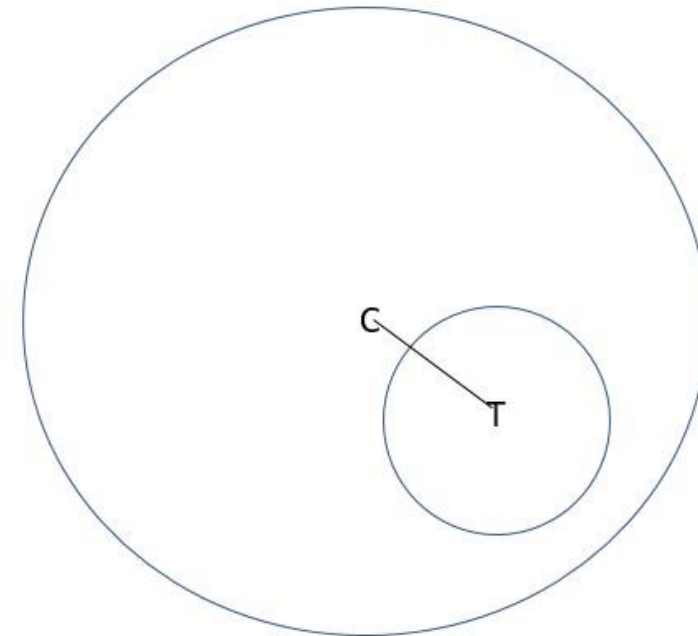
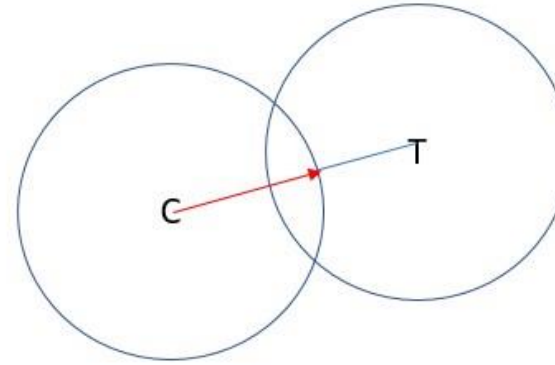
```
1: struct Gnode *Gtree;
2: currNode = buildEmptyGtree(Gtree, ranks_below, 0)
3: j = num_ranks - 1
4: while j > 0 do
5:     if currNode→assigned == -1 then
6:         if currNode→num_below ≤ 1 then
7:             Receive max and min of the leaf from rank j;
8:             currNode→this_rank = j;
9:             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
```

Read Target

- Only the master rank reads the target file named file501
- 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 datapoints" section

Search Comparison Logic

```
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);
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 (targetMagnitude < anode->maxRadius || testRadius < radius){
    caSize = 2;
    start = 0;
    end = 1;
    childArray[0] = anode->left;
    childArray[1] = anode->right;
}
```



Get Targets (Round Robin)

- For Master rank

```
sendArray = (struct data_struct *) malloc(maxSendSize * 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);  
}
```

- For other ranks

```
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);
```

Build Local Tree (Adjust Index)

```
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;
        }
    }
}
```

Local Count (Round Robin)

- Get number of datapoints within radius of target
- Same Logic as getSendArray
- Send total counts to 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)
    }
```

Global Count

- Master rank receives counts from other ranks
- Sum up all counts for a given target with a radius
- Displays final outputs

```
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);
}
```

$0 \leq \text{assigned} < \text{number of targets assigned to rank}$

$\text{radiCounts}[\text{assigned}] = [\text{id}1, \text{x}1, \text{y}2, \text{z}2, \text{id}2, \text{x}2, \text{y}2, \text{z}2, \dots, \text{id}n, \text{x}n, \text{y}n, \text{zn}]$

$0 \leq k < \text{number of targets (k represents target id)}$

$\text{allRadiCounts}[k] = [\text{x}1, \text{y}1, \text{z}1, \text{x}2, \text{y}2, \text{z}2, \dots, \text{x}n, \text{y}n, \text{zn}]$

Validation

- Used file001 to validate our code
- Used a function named 'targetTest.c' to verify if a point is within the radius of the target.
- Compared the result of targetTest.c with the result of our program for file001 and file501
- Matched the global tree leaves with the local tree heads.

Validation File 001

INDIVIDUAL COMPARES

TargetID	0.01	0.05	0.1
1	1	10	49
2	1	40	320
3	1	17	121
4	1	31	244
5	1	22	152
6	1	20	185
7	1	15	89
8	1	7	58
9	15	2085	3853
10	2	15	152
11	3	220	1276
12	1	9	74
13	1	9	67
14	8	1462	3558
15	6	2060	3561
16	48	2985	4640
17	1	76	593
18	2	300	1467
19	1	152	879
20	21	2129	4578
21	1	37	373
22	1	13	69
23	1	14	107
24	1	3	40
25	2	14	129
26	20	1558	3725
27	107	3242	3557
28	1	16	151
29	32	2313	3858
30	1	42	274
31	1	4	68
32	1	23	175
33	147	3345	3560
34	1	114	825
35	41	2558	4624
36	1	22	115
37	2	115	779
38	1	29	237
39	1	11	59
40	3	100	671
41	1	17	147

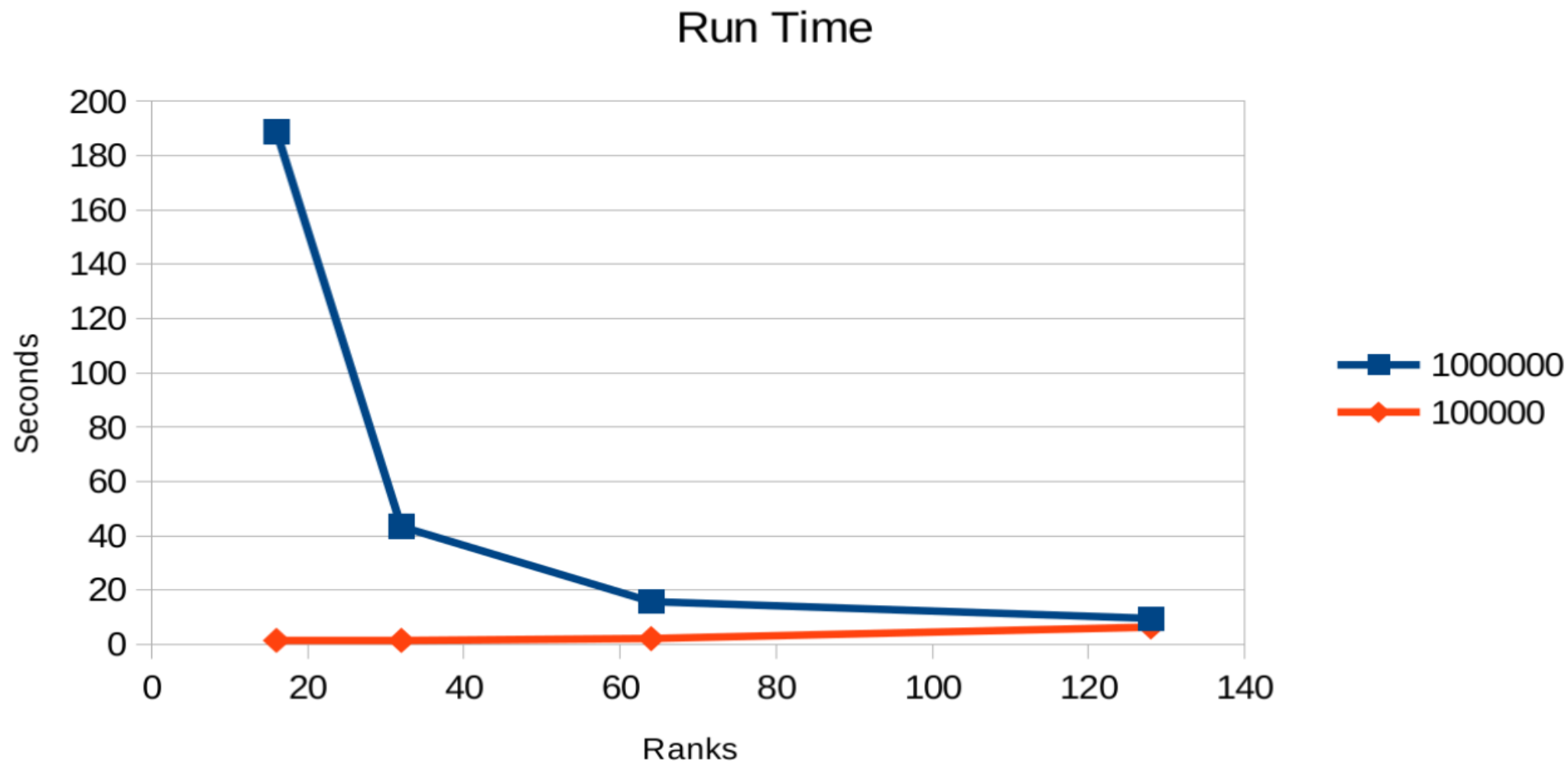
TREE SEARCH

TargetID	0.01	0.05	0.1
1	1	10	49
2	1	40	320
3	1	17	121
4	1	31	244
5	1	22	152
6	1	20	185
7	1	15	89
8	1	7	58
9	15	2085	3853
10	2	15	152
11	3	220	1276
12	1	9	74
13	1	9	67
14	8	1462	3558
15	6	2060	3561
16	48	2985	4640
17	1	76	593
18	2	300	1467
19	1	152	879
20	21	2129	4578
21	1	37	373
22	1	13	69
23	1	14	107
24	1	3	40
25	2	14	129
26	20	1558	3725
27	107	3242	3557
28	1	16	151
29	32	2313	3858
30	1	42	274
31	1	4	68
32	1	23	175
33	147	3345	3560
34	1	114	825
35	41	2558	4624
36	1	22	115
37	2	115	779
38	1	29	237
39	1	11	59
40	3	100	671
41	1	17	147

Validation file 501

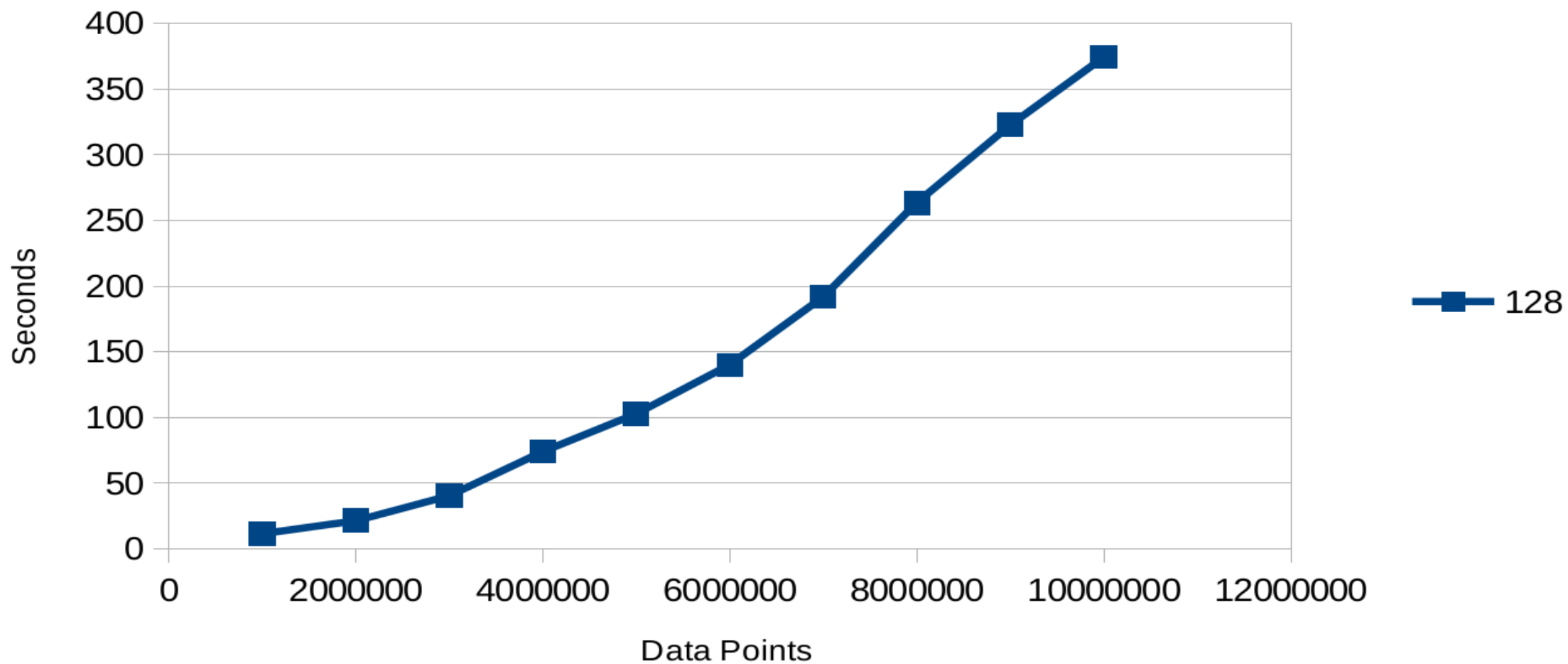
INDIVIDUAL COMPARES				TREE SEARCH			
TargetID	0.01	0.05	0.1	TargetID	0.01	0.05	0.1
10000000001	38	2715	4646	10000000001	38	2715	4646
10000000002	1	94	633	10000000002	1	94	633
10000000003	0	28	194	10000000003	0	28	194
10000000004	4	186	1131	10000000004	4	186	1131
10000000005	0	32	184	10000000005	0	32	184
10000000006	36	2625	3865	10000000006	36	2625	3865
10000000007	0	12	82	10000000007	0	12	82
10000000008	0	6	74	10000000008	0	6	74
10000000009	33	2695	4617	10000000009	33	2695	4617
10000000010	0	14	104	10000000010	0	14	104
10000000011	4	84	645	10000000011	4	84	645
10000000012	10	856	3312	10000000012	10	856	3312
10000000013	0	9	68	10000000013	0	9	68
10000000014	0	79	557	10000000014	0	79	557
10000000015	93	3250	3566	10000000015	93	3250	3566
10000000016	0	37	325	10000000016	0	37	325
10000000017	0	11	120	10000000017	0	11	120
10000000018	0	16	109	10000000018	0	16	109
10000000019	0	96	563	10000000019	0	96	563
10000000020	0	12	131	10000000020	0	12	131
10000000021	0	132	1028	10000000021	0	132	1028
10000000022	0	15	108	10000000022	0	15	108
10000000023	199	3453	3564	10000000023	199	3453	3564
10000000024	0	22	153	10000000024	0	22	153
10000000025	0	34	227	10000000025	0	34	227
10000000026	0	22	168	10000000026	0	22	168
10000000027	0	11	91	10000000027	0	11	91
10000000028	0	16	106	10000000028	0	16	106
10000000029	4	618	2893	10000000029	4	618	2893
10000000030	0	32	232	10000000030	0	32	232
10000000031	1	39	264	10000000031	1	39	264
10000000032	0	12	75	10000000032	0	12	75
10000000033	0	31	226	10000000033	0	31	226
10000000034	0	5	65	10000000034	0	5	65
10000000035	1	134	908	10000000035	1	134	908
10000000036	0	85	583	10000000036	0	85	583
10000000037	0	79	683	10000000037	0	79	683
10000000038	1	177	1122	10000000038	1	177	1122
10000000039	0	23	189	10000000039	0	23	189
10000000040	0	11	89	10000000040	0	11	89

Timing Data



Timing Data

Run time



Search Results

Table 1: Time required for 1000000 with different nodes.

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

Bottleneck

- Build Local Tree (when points is large)
- Reading data files (when points is large)
- Get Local Head (when ranks is large)
- Assigning the target (when targets is large)
- Local search (when targets is large)

Program Execution

- Python script "runTests.py" to run the different test by varying number of ranks, datapoints and target size.
- This python script creates qsub file for each run with particular target size, rank and datapoint.
- Qlogin for quick test
- Git version control.
- Each state timing data stored in .txt file
- Python code "killprocess.py" that kill the running process for all the used node for qsub job ID.

Challenges

- Splitting / Grouping MPI COMM WORLD. (Spent weeks on this)
- Print statements
- 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.

Challenges

- Split ranks
 - Without groups(no barriers)
 - With groups (with barrier)
- Comm Collections
 - With COMM(ranks<128)
 - Withouts COMS
 - Custom Collectives(unstable)
 - Split ranks(datapoints<1 milions)

Challenges

- Print Formats
- Reallocate the array

Work Distribution

- Gabriel Toban: Adjust Buckets, Round Robin, Integration
- Khem Poudel: MPI Comms, Data Read
- Toheeb Biala: MPI Functions, Local Sort

Conclusion

- Successful implementation of an orthogonal recursive bisection (ORB) of the dataset along N-dimensions with serial and parallel sort.
- Implemented for a parallel spatial binary tree that searches millions/billion points.
- Find how many points are within 3 search radii of 20 million targets .

Thank you !!!