Parallel Orthogonal Recursive Bisection

Team Metropolis:
Jamshid 'James' Farzidayeri, JJ Lay, and Graham West

COMS 7900, Capstone

Abstract

In our first project, we implemented a parallel sorting algorithm which utilized a local gradient-type optimization search to equalize the amount of data across different compute nodes in order to achieve maximum efficiency. In this project, we applied this algorithm to the problem of parallel orthogonal recursive bisection (ORB), i.e., the construction of k-d trees. In order to do this, we had to heavily modify the sorting algorithm in several ways, including 1) turning it into a callable function, 2) letting the rank 0 head node perform work while still managing the tasks, 3) incorporating the use of different MPI communicators, and 4) altering the adaptive binning technique for better convergence.

In this paper, we will discuss how our k-d tree algorithm works, how we solved the various issues plaguing parallel sort (mentioned above), and how we tested and validated our work. We conclude with a discussion of the major difficulties in completing this project and how these difficulties could be minimized in the future.

Contents

1 Introduction **2**
1.1 Workflow . . . 2
1.2 Variables and conventions . . . 2

2 Implementation **3**
2.1 `main` . . . 3
2.2 K-d tree . . . 3
2.2.1 Constants . . . 4
2.2.2 Tree Node Naming . . . 4
2.2.3 Parallel variables . . . 4
2.2.4 Serial variables . . . 5
2.2.5 Data point variables . . . 7
2.2.6 Building the tree . . . 7
2.2.6.1 `buildTree` . . . 7
2.2.6.2 `buildTree_serial` . . . 7
2.2.6.3 `buildTree_parallel` . . . 7
2.2.6.4 `getSortDim` . . . 7
2.2.7 Searching the tree . . . 7
2.2.7.1 `searchTree_serial` . . . 7
2.2.7.2 `search501` . . . 8
2.3 Altered parallel sorting . . . 8
2.3.1 `parallelSort` . . . 8
2.3.1.1 Conversion to function . . . 8
2.3.1.2 Making rank 0 do work . . . 8
2.3.1.3 Using different communicators . . . 8
2.3.2 `adaptBins` . . . 8

3 Validation **9**
3.1 Two MATLAB demos . . . 9
3.1.1 2D animation of tree building . . . 9
3.1.2 3D tree partitions from different orientations . . . 9
3.2 Other . . . 9

4 Results **11**
4.1 Timing Testing . . . 11
4.2 Search Results . . . 11

5 Conclusions **11**

1 Introduction

The purpose of this project was to create a parallel searching program by expanding on a previous parallel sorting project. Given a set of predetermined and imported data, we were to develop an orthogonal recursive bisection ORB algorithm that would facilitate in organizing the data into a KD tree. The KD tree will include information such as, daughters, parents, spacial indexing, center, and size of node. Additionally, the process should maximize the use of MPI and having multiple computing nodes performing the work. Because we will be having multiple computing nodes with MPI and eventually carry the tree into single computing nodes, the process will required both serial and parallel tree building and searching operations. The goal is to have an extremely efficient method of searching through the data. Eventually a search using three radii and twenty million points from an existing file will be performed and that output will be saved to a file for later comparison.

1.1 Workflow

One of the major issues we encountered during our last project was overwriting each other's GitHub submissions. Our resolution was to create a master branch and three sub branches. Each person was assigned a sub branch that they could modify as they pleased. However, a modification in the master branch parallel approved folder required two or more party members consent. This vastly reduced the amount of issues encountered during push/pull request.

Another adjustment we made for our development process was to write our code together and in an accommodating space as opposed to assigning individual sections of code. We met on a regular basis, generally starting during our scheduled class period and extending through the lunch period. This allowed us to catch errors quickly, avoid merge conflicts and most significantly improved everyone's knowledge of how each component of the code works. When the project was assigned each team member prototypes a version of serial KD tree in their preferred language. After several meetings it was determined that Graham's MatLab implementation would be used as the guide for the project.

1.2 Variables and conventions

Note that we distinguish between **tree** nodes and **compute** nodes to avoid confusion.

Counts:

- N: number of nodes
- M: max number of allowed time steps
- L: number of lines to read per file
- L_w: number of lines on the wth worker
- D: total number of lines/data points

Indices:

- $m = 0, \cdots, M$: the time step of the bin adaptation scheme (likely less than M)
- $n = 0, \cdots, W$: spans the nodes
- $i = 0, \cdots, N$: spans the bin edges/indices
- $j = 0, \cdots, N-1$: spans the bin counts (this will occasionally subscript binI/E as well)
- $\ell_w = 0, \cdots, L_n - 1$: spans the lines on the n-th node
- $k = 0, \cdots, 3$: the data column being sorted

Variables:

- $\text{data}^n_{4\ell+k}$: data point on ℓth line and kth column on the n-th node
- E^m_j: bin edges (0 indexed) at time step m
- $I^{n,m}_j$: bin indices on node n
- $C^{n,m}_j$: bin counts on node n at time step m
- C^m_j: total bin counts on head node (sum of node C's) at time step m

2 Implementation

Here we discuss our implementation of the code
 we used C++ with C MPI calls, mpirun, qsub/qlogin, valgrind

2.1 `main`

Our `main` program is divided into three main phases (each of which is comprised of multiple functions): data import, tree building, and tree searching. The data import phase collects all of the relevant filenames of the data files, reads them, and places them into a single 1D array. The tree building phase begins by initializing an empty Tree struct (see below for explanation of the struct) which is then passed into the `buildTree` function which alters the struct's contents. The entire tree can then be navigated by using the tree's left, right, and parent fields. The final tree searching phase takes the completed tree struct as an argument along with a search sphere. The number of points within the search sphere is calculated for each

There are four variables which govern the performance of the KD tree and the size of the problem. First are the number of data files to read and the number of lines to read per file. These values are set in the beginning of `main` and they have a maximum value of 500 and 20,000,000, respectively. Third, is the number of compute nodes. In general, increasing this number should speed up execution of the program. Its value is set in the command line when running the program and its value must be smaller than the number of files. Last, is the number of lines to read from the 501-st data file. These are the centers of the search spheres. The value is set in `search501`.

2.2 K-d tree

The design of the k-d tree required storing information for organizing the parallel distribution of the data and the data held locally on the node. Rather than create two different type of structures, the code utilized a single C `struct` which is defined as:

```
struct Tree {
    // Pointers used for local tree
   Tree *p;    // Parent
   Tree *l;    // Left child
   Tree *r;    // Right child
   int i;      // Sort dimension used to split this node
   int source; // Which buildTree function created it

    // Pointers used for parallel tree
   MPI_Comm parentComm;  // Parent communicator
```

```
    MPI_Comm leftComm;     // Left child communicator
    MPI_Comm rightComm;    // Right child communicator
    MPI_Comm thisComm;     // Communicator that the node belongs to

     // Each node in the tree has a unique identifier
    string name;

     // For nodes with children, this holds the min and max for
     // the child nodes
    double x1;  // Min x
    double x2;  // Max x
    double y1;  // Min y
    double y2;  // Max y
    double z1;  // Min z
    double z2;  // Max z

    int depth;  // Depth of the node in the tree
    int n;      // Number of points

    double c[4];    // Center of this tree
    double radius;  // Radius of this tree

    double d[4];   // Data point coordinates
    double index;  // File row index from source data
}
```

2.2.1 Constants

C preprocessor identifiers were defined and used in place of numeric constants for readability. Data points were defined as an array of `double` of size four consistently even when only three values were needed. The four positions were referenced using:

```
#define _INDEX_   0  // Row index from source datafile
#define _X_       1
#define _Y_       2
#define _Z_       3
```

A single tree is create and populated by both the parallel and serial versions of `buildTree`. To aid with debugging, each node is marked with an identifier to show which function created it. The identifiers for the sources are:

```
#define _Source_buildTree_unknown     0
#define _Source_buildTree_parallel   -1
#define _Source_buildTree_serial     -2
```

2.2.2 Tree Node Naming

Each node in the tree is assigned a unique identifier determined by its position. The top node is named `t`. The two child nodes of `t` are `tl` and `tr`. An `l` or `r` is appended to the parent's name to designate the direction it is below the parent. The final node of the parallel section appends an `*` to designate the transition from parallel to serial. The node of the tree containing the data point appends an `!`.

2.2.3 Parallel variables

The `struct` variables used during the initial division of the tree among the nodes and their purpose are:

`i` The dimension that was split by the MPI communicator.

`source` During the parallel phase, this value is set to `_Source_buildTree_parallel`.

`parentComm` The parent MPI communicator that created this node. The root node stores a value of `MPI_COMM_SELF` in lieu of a null.

`leftComm` The MPI communicator of the *left* child tree for the current node.

`rightComm` The MPI communicator of the *right* child tree for the current node.

`thisComm` The MPI communicator of the current compute node.

`name` The unique name of the tree node.

`x1` The minimum value of x stored in this subtree.

`x2` The maximum value of x stored in this subtree.

`y1` The minimum value of y stored in this subtree.

`y2` The maximum value of y stored in this subtree.

`z1` The minimum value of z stored in this subtree.

`z2` The maximum value of z stored in this subtree.

`depth` The depth within the tree of the current node. The top node has a depth of zero.

`n` The number of points contained within the subtree where the current node is the root.

`c[4]` The center of the subtree. Index for the array is described in section 2.2.1.

`radius` The distance from the center of the subtree to the furthest point.

2.2.4 Serial variables

`p` Pointer to the parent of the current node. This value is `null` for the top node.

`l` Pointer to the *left* child of the current node. This value is `null` for the data point.

`r` Pointer to the *right* child of the current node. This value is `null` for the data point.

`i` The dimension that was split during the local tree construction.

`source` During the serial phase, this value is set to `_Source_buildTree_serial`.

`name` The unique name of the node.

`x1` The minimum value of x stored in this subtree.

`x2` The maximum value of x stored in this subtree.

`y1` The minimum value of y stored in this subtree.

`y2` The maximum value of y stored in this subtree.

`z1` The minimum value of z stored in this subtree.

`z2` The maximum value of z stored in this subtree.

`depth` The depth within the tree of the current node. The top node has a depth of zero.

`n` The number of points contained within the subtree where the current node is the root.

`c[4]` The center of the subtree. Index for the array is described in section 2.2.1.

`radius` The distance from the center of the subtree to the furthest point.

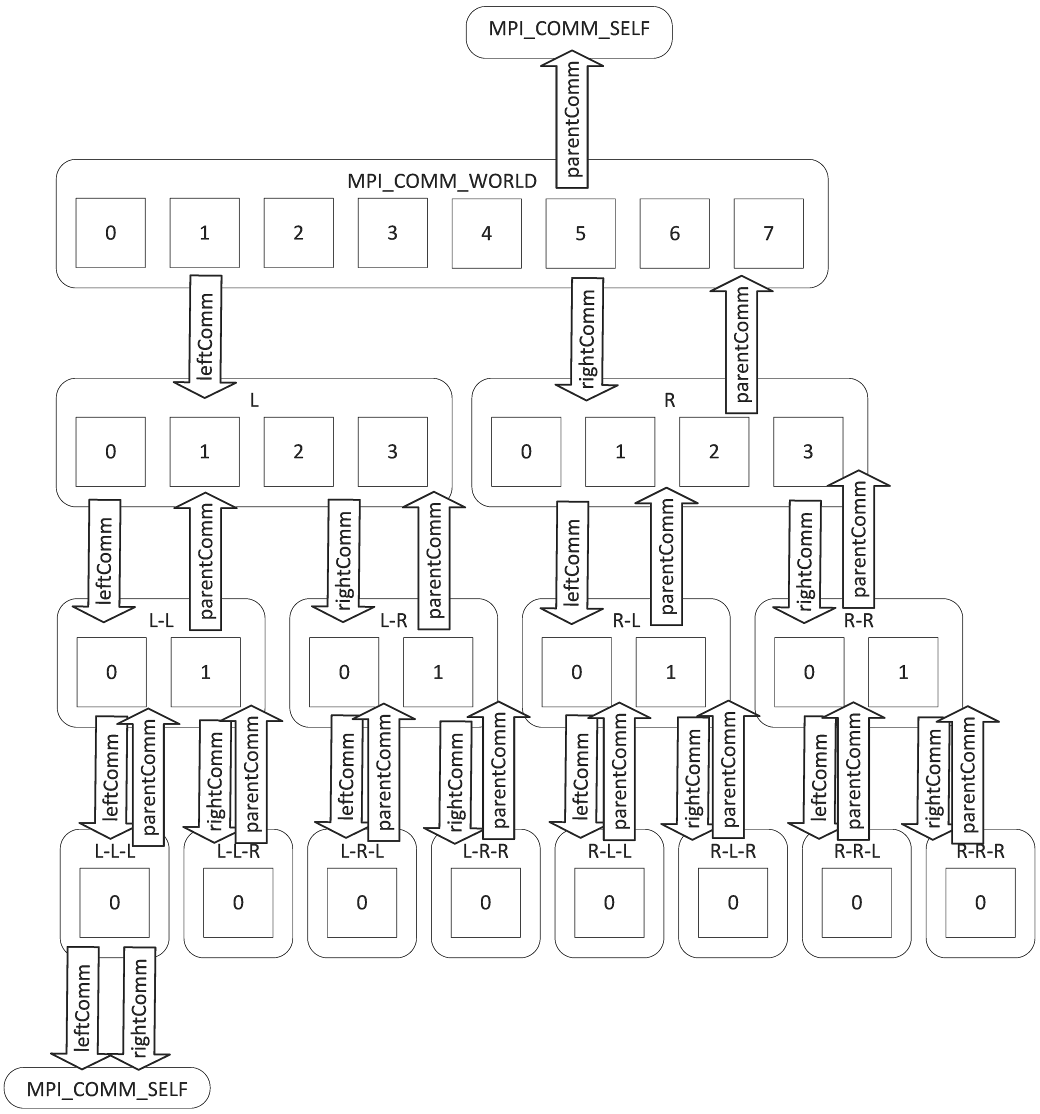

Figure 1: Example of parallel variables using eight nodes

2.2.5 Data point variables

`p` Pointer to the parent of the current node.

`l` Pointer to the *left* child of the current node. This value is `null` for the data point.

`r` Pointer to the *right* child of the current node. This value is `null` for the data point.

`source` During the serial phase, this value is set to `_Source_buildTree_serial`.

`name` The unique name of the node.

`depth` The depth within the tree of the current node. The top node has a depth of zero.

`n` The number of points contained within the subtree where the current node is the root.

`d[4]` The data point.

`index` The row index of the data point as read from the source file.

2.2.6 Building the tree

2.2.6.1 `buildTree` This function gets the number of compute nodes q available in the current communicator and determines which function to run. If $q > 1$, then we can still do a parallel sort with 2 compute nodes, so `buildTree_parallel` is entered. If $q = 1$, then `buildTree_serial` is entered.

2.2.6.2 `buildTree_serial` This was the first function written after our initial prototyping phase was completed. It essentially performs a serial version of ORB which can be executed on a single compute node.

Upon completion, `buildTree_serial` calls itself instead of `buildTree` since we still have $q = 1$.

2.2.6.3 `buildTree_parallel` This function performs essentially the same tasks as `buildTree_serial`, but utilizing multiple nodes for speedup. Specifically, it takes advantage of `parallelSort` in order to speed up the partitioning of the data along the longest axis (determine by `getSortDim`, discussed below). After the data has been sorted, the data is split by placing lower half (w.r.t. the sorted data values) of the compute nodes into a left communicator and the upper half into a right communicator. Each half then calls `buildTree`.

2.2.6.4 `getSortDim` This function is used by `buildTree_parallel` in order to determine which is the longest axis, i.e., the sort dimension. In addition to this, it also fills in the struct for the current tree node with the global min/max in all three dimensions. To do this, each node gets its own min/max for each axis and sends them to rank 0 (w.r.t. the current communicator). Rank 0 then determines the global min/max and them MPI_Bcast's those values back to the other ranks. This allows each communicator to sort their data independently along different axes. The center of the bounding box defined by these values min's/max's is also stored in the tree struct.

2.2.7 Searching the tree

2.2.7.1 `searchTree_serial` This function is called by all ranks and returns an integer which equals the number of points found by the rank that called it. It takes as arguments the point and radius (which specifies a search sphere) and the root of the tree created by `buildTree`. To perform the search, a check is done to determine of the search sphere intersects the bounding sphere of each data partition:

$$r^2_{\text{sphere center to box center}} \leq (r_{sphere} + r_{box})^2 \tag{1}$$

We used the squared version of the formula to avoid computing the square root, thus saving time. If this check returns false, then there are no data points within the search sphere contained in that partition and the function exits. If true, then another check is performed to check if the left or right child tree are NULL. If they are both NULL, then by our definition of the tree, we have found a point that is a leaf of the tree, thus we increment the count and exit the function. Otherwise, we recursively call the function again using the correct child tree.

2.2.7.2 `search501` Each compute node calls this function which reads the 501-st data file, whose contents are the centers of the search spheres. It then loops `searchTree_serial` for all of the sphere centers for three different radii (0.01, 0.05, 0.10). The counts on each compute node are then stored. After all searches are complete, an MPI_Reduce is performed by all nodes, thus adding all of the counts into a single array which is exported to a file.

2.3 Altered parallel sorting

We had to adjust our parallel sorting algorithm a great extend to integrate it into the new project.

2.3.1 `parallelSort`

2.3.1.1 Conversion to function One of our first obstacles was modifying our original `parallelSort` program to work for this project. The first issue was having `parallelSort` operate as a function. We accomplished this by removing the MPI setup and Data Import sections from `parallelSort` and placing them into a new Main. While doing this we encountered issues with how the data was passed using both a pointer and array notation. Because several functions within the `parallelSort` function would need to change both the number of rows and the data we had to pass as pointers and pointers to arrays.

2.3.1.2 Making rank 0 do work The second issue was our use of rank 0 which as the manager in our original `parallelSort` program. Our implantation of `parallelSort` had rank 0 manage while all the other ranks were compute nodes which contained and performed operations on the actual data. However, when utilizing a KD tree all of the ranks are needed as compute nodes otherwise each time the tree forked and a new communication branch was formed the process would lose a rank due to a new rank 0 being formed. To resolve this problem we had to make several clever adjustments in the sections of the code, most notably the adapt binning section. Other sections just needed to have the iteration increment from 0 instead of 1.

2.3.1.3 Using different communicators Since kdtree requires multiple comms, it was necessary that `parallelSort` (any many functions it contains) was given the current communicator. This was not a terribly difficult modification, but it was quite tedious since there were many functions which required alterations to their header files.

2.3.2 `adaptBins`

Although our original adaptive binning scheme performed well, we wanted something which could do even better since *k*-d trees require many parallelSort calls. As a review, here is our original method:

$$\begin{aligned} \Delta C &= 2.0(C_{i+1}^m - C_i^m)/(C_{i+1}^m + C_i^m) \\ \Delta E &= E_{i+1}^m - E_i^m \\ E_i^{m+1} &= E_i^m + \alpha \Delta C \Delta E \end{aligned} \tag{2}$$

where C are the bin counts, E are the bin edges, and $\alpha < 0.5$. This method will occasionally devolve into oscillatory behavior and not converge to the correct value. To combat this in the *k*-d tree project, we added a scale factor S which decreases over time:

$$\begin{aligned} S(m) &= 1 - (1 - 0.1)(1 - \exp(-0.03m) \\ E_i^{m+1} &= E_i^m + \alpha S(m) \Delta C \Delta E \end{aligned} \tag{3}$$

Since this method is local, it converges slowly at bin edges far away from high-density clusters. As such, we attempted to replace this method with a global method which uses linear interpolation to estimate where the bins would be evenly distributed. Define the function $\hat{C}(x)$ as the linear approximation of the cumulative count distribution of the data points (so that it normalizes to the number of data points, not unity). Then,

$$\hat{C}(x) = \hat{C}(E_{i'}^m) + C_{i'}^m \frac{x - E_{i'}^m}{E_{i'+1}^m - E_{i'}^m} = (i+1)\frac{D}{N} \tag{4}$$

where i' is the maximum index such that $\hat{C}(E_{i'}) < (i+1)\frac{D}{N}$ (note that i' and i are distinct integers). The right equality implies that we should solve for the value of x such that it holds. This x value will be the new i-th bin edge, E_i^{m+1}. Therefore,

$$E_i^{m+1} = E_{i'}^m + \Big((i+1)\frac{D}{N} - C(E_{i'}^m)\Big)(E_{i'+1}^m - E_{i'}^m)/C_{i'}^m \tag{5}$$

We discovered that this adaptation technique performs very will in initial steps of adaptation, but is prone to oscillations near clusters of points. Now, since each of the two methods perform better and worse in different contexts, are solution was to simply alternate between at each step. This solved all of our convergence issues in testing. (Note that we still use the same binary search-based binning technique and stopping criterion from the previous project.)

3 Validation

3.1 Two MATLAB demos

During our development of the code we wanted a way to verify the process was being performed accurately. Therefore, after the parallel side of the KD tree was complete we had our C++ code output the min and maxes of the X, Y, and Z columns and save those outputs to a file. We then wrote MATLAB code that would read in those files and plot a box that encapsulated the three values. This figure represents the boundaries of 25 nodes for a sample of data. As can be seen from this image, each of the bounding boxes vary in limiting size, however all the boxes contain a relatively equal number of points. That number can be verified through our parallel sorting operations.

3.1.1 2D animation of tree building

When prototyping the code in MATLAB, we generated animated visualizations of the construction of a 2D tree. Due to the format, we cannot include the animation within this paper, but it shall be included in the presentation.

3.1.2 3D tree partitions from different orientations

Another testing method we performed during our development was validation through variation of input parameters. We held constant the number of data points read, the searching point and search radius, but varied the number of nodes that were used in the process. Then we had each node output the number of data points within their tree that fell within the search point and radius. Although the number of points per tree would vary depending upon the number of nodes used, the sum of all those points would remain constant. From this test we confirmed that the search tree was working as intended.

3.2 Other

During the testing process we also varied the searching parameter radius in two ways. First, we made the search radius very large and as expected we encapsulated all points. Second, we reduced the search radius and the search point so that it should only include a single location and our results were perfect.

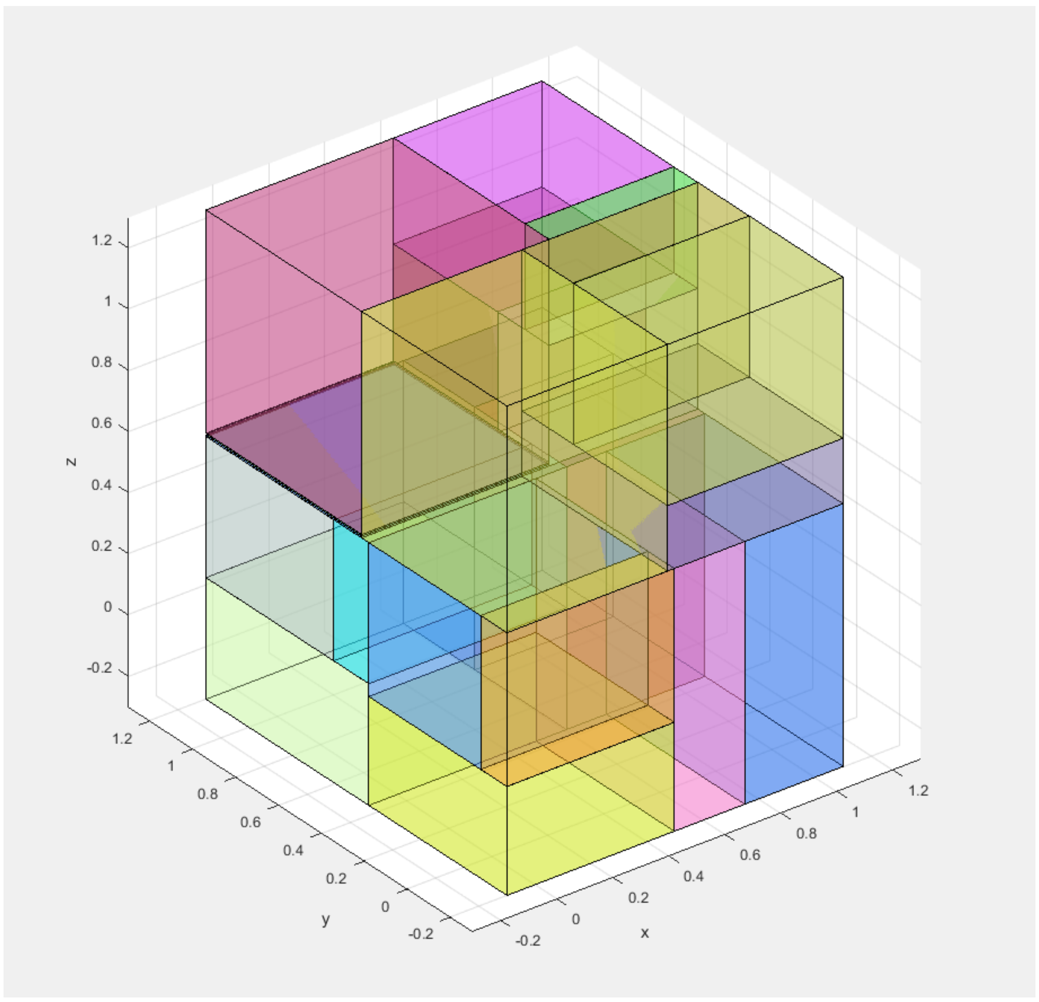

Figure 2: Example of KD tree partitions

4 Results

4.1 Timing Testing

4.2 Search Results

5 Conclusions

This project was an excellent test for managing MPI in the context of large amounts of data and long run times. It also provided an opportunity to work as a well-functioning team and develop a synergistic group dynamic. We were able to overcome significant setbacks. Below is a summary of different aspects of our performance, including some problems which we struggled with as well as our successes and possible future work.

Problems:

- `parallelSort` conversions
- memory leaks
- multiple communicators (comm)
- `adaptBins` convergence problems
- debug print statement clutter
- array out of bounds issues
- inconsistent usage pointer-to-pointer calls for *data[] and *rows (due to `swapArrayParts`)
- no planning for function arguments and return values (constant editing of h-files)
- testing was difficult due to cluster overloading and hardware errors

Successes:

- few merge conflicts and fast debugging through extreme coding and Git branches
- visualizing output through MATLAB
- efficient delegation of tasks

Future work:

- cloud computing
- use of coding techniques for personal research