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

The purpose of this project was to create a parallel searching algorithm by expanding on a previous parallel sorting project. Given a very large set of data (10^{10} points), we developed an orthogonal recursive bisection (ORB) algorithm which organizes the data into a k-d tree. The k-d tree includes information such as daughter and parent nodes, spacial indexing and ranges, etc. Additionally, the implementation also maximized the use of MPI, having multiple computing nodes performing the work. Due to the recursive nature of the problem as well as our implementation of it, we required both serial and parallel tree-building functions. Eventually a search using three radii and 2×10^7 points from an existing file were performed and the output saved to a file.

1.1 Workflow

One of the major issues we encountered during the previous project was overwriting each others' 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 required two or more team members' consent. This vastly reduced the amount of issues encountered during push/pull request.

Another adjustment we made for our development process was to use so-called "extreme coding." In this context, only one team member actually writes/runs code while the other team members help plan, troubleshoot, debug, etc. We would meet in WPS 305 as opposed to the computer lab, due to the lack of noise, larger screens, and number of whiteboards. REgarding the coding itself, Graham was typically the coder with James watching and reviewing what was written and ran. JJ would usually have a second terminal open so that he could do small tests or refer to certain code sections (or Google search troublesome issues) when we had questions. We met on a regular basis, generally starting during our scheduled class period and extending through the lunch period. In summary, this technique allowed us to find errors quickly, debug quickly, avoid merge conflicts, and improved everyone's knowledge of how each component of the code operates.

Also, when the project was assigned, each team member prototyped their own version of a serial k-d tree in their preferred language. After a discussion on the merits of each, 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

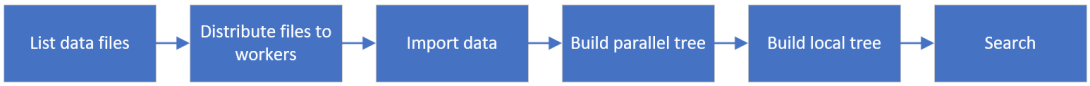
Indices:

- $m = 0, \cdots, M$: 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



2.1 Development

The application was developed in a mixture of C and C++ on a Linux cluster. The compiler for the project was `g++` version 4.8.5-28 with the switch `-std=c++11` in order to utilize features introduced with that standard. The three most commonly used C++11 standards were placeholder type specifiers (i.e., `auto`), range-based `for` loops, and the `nullptr` constant.

The C interface to MPI was selected for its familiarity and general support. While there are object-oriented C++ libraries such as `Boost.MPI` the team elected to use the C version as they had experience with that environment and time to learn a new library was a limiting factor. Another concern was the installation of alternative libraries beyond Open MPI 2.1.1.

Debugging was performed using the tools `gdb` and `valgrind`. `gdb` was used primarily while developing the serial code for building the tree to identify the source of segmentation faults. `valgrind` was instrumental in locating memory leaks and illegal memory references during the parallel development. Code was executed with the command:

```
mpirun -np 4 valgrind ./main > dump.txt 2> dump2.txt
```

One major challenge during the debugging process was that output was not displayed in chronological order making it difficult to understand the flow of the code. A solution we implemented was to print a numeric identifier during the execution at the beginning of each line. Each section of code was assigned a five digit numeric value, and the `mpi` rank was printed along with this (an additional output sorting technique was use of `sleep(myRank)`):

```
39300 : receiveFiles: Rank 93 received /home/hal2a/localstorage/public/
                              coms7900-data/datafile00122.txt as tag 0
39301 : receiveFiles: Rank 93 received DONE! as tag 1
35400 : receiveFiles: Rank 54 received /home/hal2a/localstorage/public/
```

```
                              coms7900-data/datafile00157.txt as tag 0
35401 : receiveFiles: Rank 54 received /home/hal2a/localstorage/public/
                              coms7900-data/datafile00023.txt as tag 1
35402 : receiveFiles: Rank 54 received DONE! as tag 2
30000 : receiveFiles: Rank 0 received /home/hal2a/localstorage/public/
                              coms7900-data/datafile00407.txt as tag 0
39901 : receiveFiles: Rank 99 received DONE! as tag 1
30001 : receiveFiles: Rank 0 received /home/hal2a/localstorage/public/
                             coms7900-data/datafile00361.txt as tag 1
30002 : receiveFiles: Rank 0 received DONE! as tag 2
30900 : receiveFiles: Rank 9 received /home/hal2a/localstorage/public/
                             coms7900-data/datafile00020.txt as tag 0
30901 : receiveFiles: Rank 9 received /home/hal2a/localstorage/public/
                             coms7900-data/datafile00046.txt as tag 1
30902 : receiveFiles: Rank 9 received DONE! as tag 2
45700 : receiveFiles: Rank 157 received /home/hal2a/localstorage/public/
                               coms7900-data/datafile00086.txt as tag 0
45701 : receiveFiles: Rank 157 received DONE! as tag 1
70000 : Rank 11 is calling buildTree
70000 : Rank 70000 : Rank 125 is calling buildTree
70000 : Rank 131 is calling buildTree
```

The output was then sorted using:

```
cat dump.txt | sort | less
```

Code was executed using both interactively by logging into a compute node using `qlogin` and as a batch job using `qsub`. Short runs of a few minutes were performed interactively when failures occurred quickly or to test a new section of code on smaller data sets. Longer runs that needed more than eight nodes were submitted to the Sun Grid Engine batch system.

2.2 `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 a list of search spheres and radii.

There are four variables which govern the performance of the k-d tree and the size of the problem. First and second are the number of data files and the number of lines per file to read. 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 let's us increase the problem size. 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`.

2.2.1 `listFiles`

The `listFiles` function has the following definition:

```
vector<string> listFiles(string path, int numFiles);
```

It accepts two parameters:

`path`	The path to the data files to be used
`numFiles`	The maximum number of files to be used by the program

The return value is a Standard Template Library `vector` containing strings with the filenames. The vector has a maximum length of `numFiles`. Files are not gathered in alphabetical order but in the order returned from the operating system. For example, for `numFiles = 10`, it is unlikely that the files returned will be `datafile00001.txt` through `datafile00010.txt`.

2.2.2 `distributeFiles`

The `distributeFiles` function is defined as:

```
void distributeFiles(vector<string> files, int numWorkers);
```

It accepts two parameters:

`files`	The list of files returned by `listFiles`
`numWorkers`	The number of nodes allocated for the job

The function does not return any values to the caller. It iterates over the list of files and sends them to the workers in a round-robin method to ensure that the data is distributed as evenly as possible without knowing the true length of each data file. Files are sent asynchronously since rank 0 participates in the construction of the tree and searching.

2.2.3 `receiveFiles`

The `receiveFiles` function is defined as:

```
vector<string> receiveFiles(int myRank);
```

It accepts a single parameter:

`myRank`	The MPI rank of the current process

`receiveFiles` performs an `MPI_Receive` to obtain a list of files sent by `distributeFiles`. These are stored in an STL `vector` and returned to the calling function.

2.2.4 `importFiles`

The `importFiles` function is defined as:

```
void importFiles(vector<string> files, int myRank,
   float *myData, int *rows, int *cols,
   int maxRowsPerFile, unsigned long int arrayLimit);
```

This function accepts seven parameters:

`files`	An STL vector of files from `receiveFiles`
`myRank`	The MPI rank of the current process
`myData`	A one-dimensional array in which to store the data from the files
`rows`	A pointer to a variable to store the number of rows imported
`cols`	A pointer to a variable to store the number of columns imported
`maxRowsPerFile`	The maximum number of rows to read from each file
`arrayLimit`	The maximum size of the allocated array

The `importFiles` function iterates over the list of files received and opens each for reading. It uses `fscanf` to read each line directly into the array pointed to by `myData` in order to minimize data movement. The file is closed when the end of the file is reached, `maxRowsPerFile` is met, or the total data read from all files equals `arrayLimit`. The number of rows is written to `rows` and the number of columns to `cols`. The variable `myRank` is passed for debugging purposes. Results are returned through the references passed as variables.

2.3 k-d tree

The design of the k-d tree required storing information for organizing both 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`. This `struct` underwent a massive overhaul/trimming near the end of the project in an attempt to decrease to total memory usage. Initially, the `struct` was 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
}
```

This `struct` contains multiple fields which either were only used for debugging or not even used at all. Clearly, this structure proved to be too bloated when attempting to import all 10 billion rows of data as well as the 20 million rows of search rows. The so-called "trimmed" tree `struct` is defined as:

```
struct Tree {
   Tree *p;  // Parent
   Tree *l;  // Left child
   Tree *r;  // Right child

   MPI_Comm parentComm, leftComm, rightComm, thisComm;

   float x1;  // Min x
   float x2;  // Max x
   float y1;  // Min y
   float y2;  // Max y
   float z1;  // Min z
   float z2;  // Max z

   float c[4];  // Center of this tree
   float radius;

   float d[4];  // Data point
}
```

As can be seen, we made an additional change by changing all `double` data types to `float` types in order to conserve memory. This change affected every portion of the code, so we performed a global find and replace on all `.cpp, .h` files in our repository using the following:

```
find ./ -type f -exec sed -i 's/DOUBLE/FLOAT/g' {} \;
find ./ -type f -exec sed -i 's/double/float/g' {} \;
```

This change only required us to alter the `fscanf` statement to read `float`'s as opposed to `long float`'s.

During the parallel construction phase, nodes are separated into different MPI communicators as shown in Figure 1. The four `MPI_Comm` variables store the references to the parent, left and right children, and current communicators. The value `MPI_COMM_SELF` is used as a "null" value for these pointers to indicate no communicators exist in the given direction.

Once the nodes became members of a communicator containing only itself, `buildTree_serial` constructed the local serial tree using the same structure but using the `Tree` pointers rather than the `MPI_Comm` variables.

2.3.1 Constants

C preprocessor identifiers were defined and used in place of numeric constants for readability. Data points were defined as an array of `float`'s 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 created 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
```

2.3.2 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 `!`. This was one of the fields which was trimmed away after debugging was completed.

2.3.3 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 compute node.
`rightComm`	The MPI communicator of the *right* child tree for the current compute 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 2.3.1.
`radius`	The distance from the center of the subtree to the furthest point.

2.3.4 Serial variables

The `struct` variables used during the final division of the local serial tree on the compute nodes and their purpose are:

`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 2.3.1.
`radius`	The distance from the center of the subtree to the furthest point.

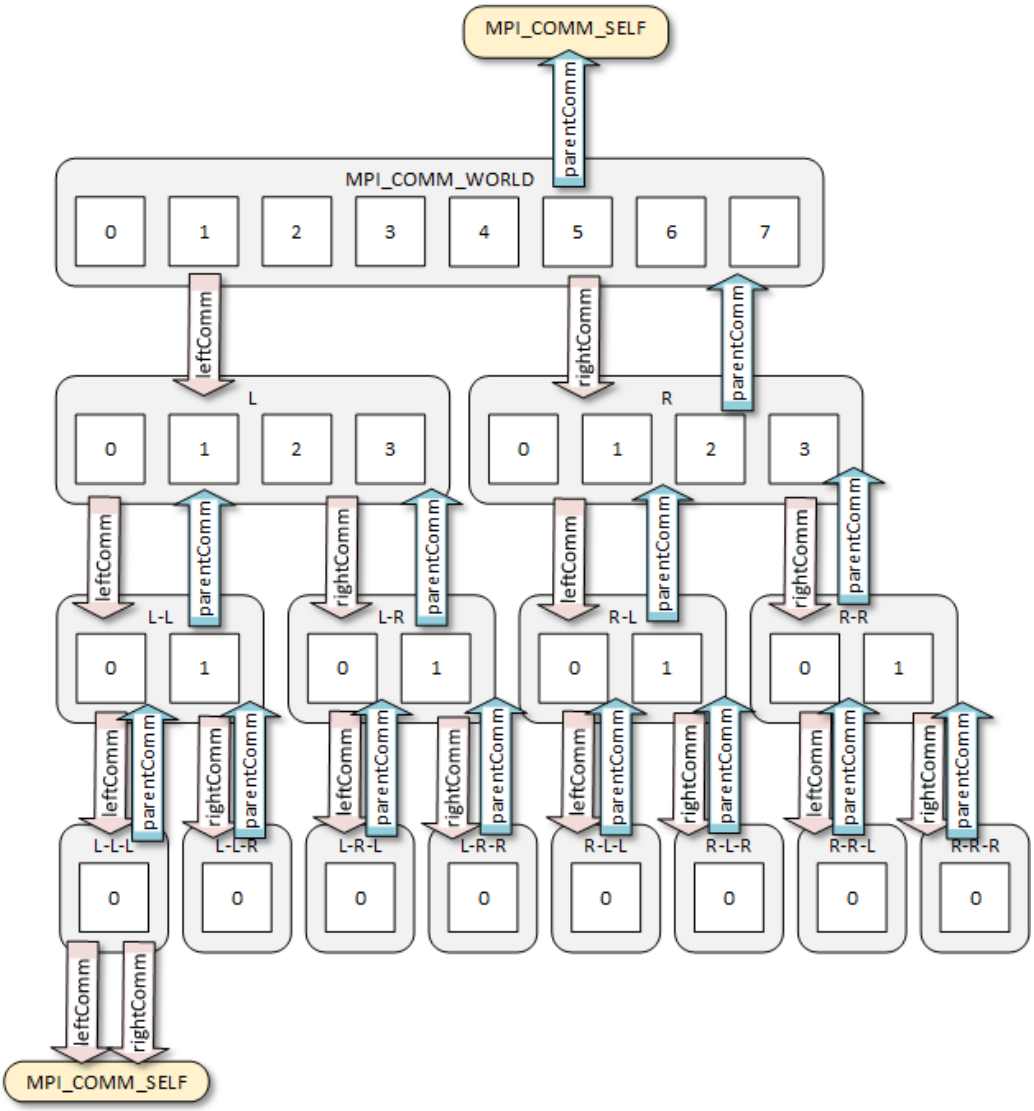


Figure 1: Example of parallel variables using eight nodes

2.3.5 Data point variables

The `struct` variables used for storing the data point within the tree and their purpose are:

`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.

2.3.6 Building the tree

2.3.6.1 `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 at least two compute nodes, so `buildTree_parallel` is entered. If $q = 1$, then `buildTree_serial` is entered.

2.3.6.2 `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` recursively calls itself instead of `buildTree` since we still have $q = 1$.

2.3.6.3 `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`.

2.3.6.4 `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.

2.3.7 Searching the tree

2.3.7.1 `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.

2.3.7.2 `search501` Each compute node calls this function which reads the data file `datafile00501.txt` whose contents are the centers of the search spheres. The function uses `importFiles` to read the file and passes the number of search rows in `maxRowsPerFile`. 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 sent to `stdout`. When the job is submitted using `qsub`, the output is directed to a file with the name of the job.

2.4 Altered parallel sorting

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

2.4.1 `parallelSort`

2.4.1.1 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.

2.4.1.2 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.

2.4.1.3 Using different communicators Since k-d tree 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.

2.4.2 `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 $\hat{C}(E_{i'}) < (i+1)\dfrac{D}{N}$ (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,

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

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 k-d tree was complete we had our C++ code

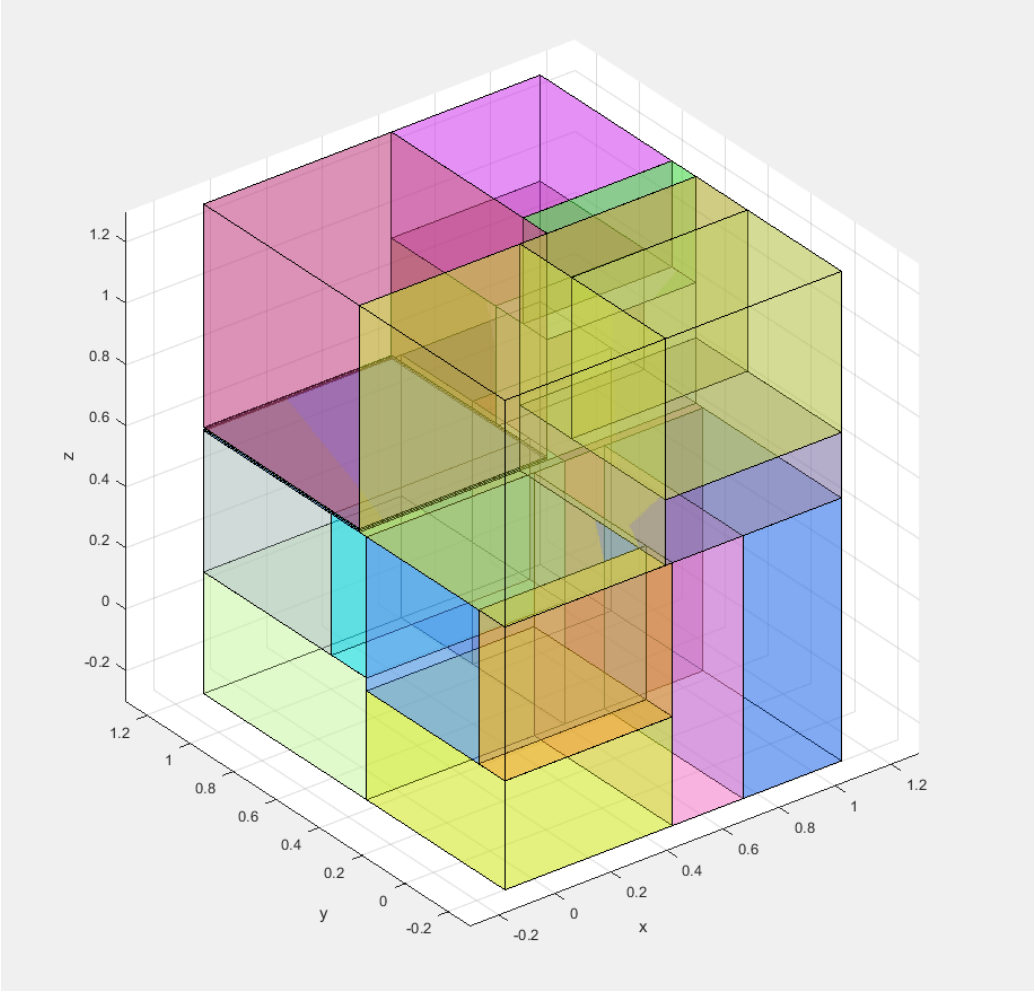


Figure 2: Example of k-d tree partitions

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.

4 Results

4.1 Timing Testing

The timing data collected does not provide a clear understanding of the scalability of the algorithms. Figure 3 displays the performance data collected for varying values of nodes and data rows. The height of the chart represents the time to execute the program. The x-axis is `log(TotalLines)`, the z-axis is the number of nodes the program was executed on, and the y-axis is the `log(Runtime)`.

The upper left of the plane is truncated due to the inability to execute the program for large datasets on fewer nodes. 20% of the timing data had to be excluded due to varying multiple parameters simultaneously. The original plan had been to perform a fully-nested Analysis of Variance (ANOVA) test on the data to understand which parameters impacted performanced the greatest. However, due to the programming challenges encountered and cluster utilization, this did not occur. Another cause of the irregular performance is the sharing of the cluster. Teams would run tests using `qlogin` (including Team Metropolis) rather than submitting them through `qsub` to allow the job scheduler.

A subset of the performance data is presented in the following table.

	Execution times (seconds)			
Total Lines	32 Cores	64 Cores	128 Cores	200 Cores
50,000	0.55	0.57	15.49	15.45
500,000	4.18	5.51	4.06	9.18
5,000,0000	39.49	50.01		65.05
50,000,000	234.73	404.49		574.71
500,000,000		2,094.11	3,685.35	
1,000,000,000		4,146.69		
2,000,000,000				21,355.30

The execution time of the program increases as more nodes are added while keeping the total data rows constant. One possible explanation for this is that on smaller numbers of nodes the job was not running on systems occupied by other teams. However, as the number of nodes increased, the job began contending for resources with other jobs.

Figure 4 shows the runtimes for four different job sizes (32, 64, 128, and 200 nodes) with varying numbers of data rows from 50,000 to 50,000,000. Figure 5 shows the runtimes from 50,000 to 2,000,000,000 (the largest job size that completed successfully).

Figure 6 provides a glimpse into the performance of the various functions of the program.

Function	**Runtime**	**Percent**	**Note**
ListFiles	0.001 seconds	0.002%	Lists contents of data directory
DistributeFiles	0.001 seconds	0.002%	Distributes filenames round-robin to nodes
ReceiveFiles	0.218 seconds	0.386%	Receives list of files to process
ImportFiles	20.889 seconds	36.976%	Reads assigned text files into memory
BuildTree	34.353 seconds	60.808%	Builds parallel and serial trees. Includes data swapping among nodes.
SearchTree	1.032 seconds	1.827%	Searches the constructed tree and produces results
Total	**56.494 seconds**	**100.000%**	

4.2 Search Results

The program exports a table at the end of the run displaying the number of points contained in each of three radii centered at each row of points from `datafile00501.txt`. The first twenty rows

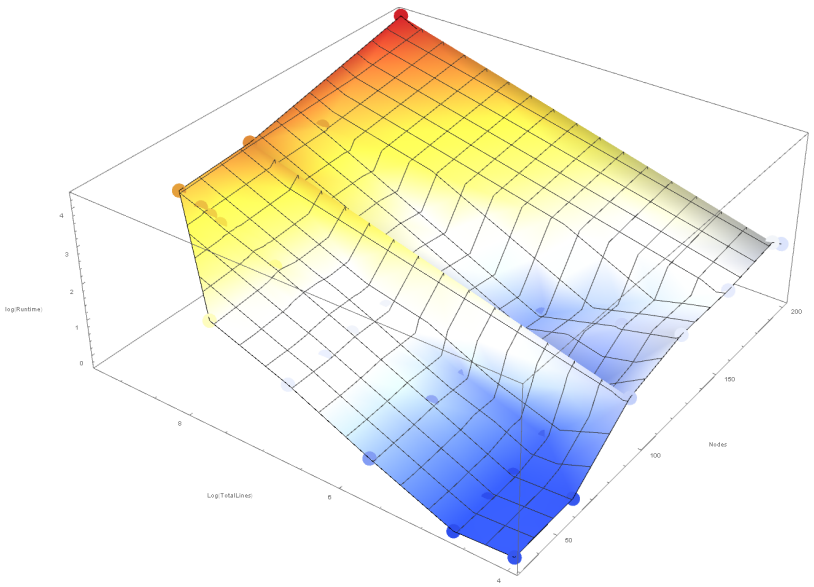


Figure 3: Execution time versus nodes and total lines

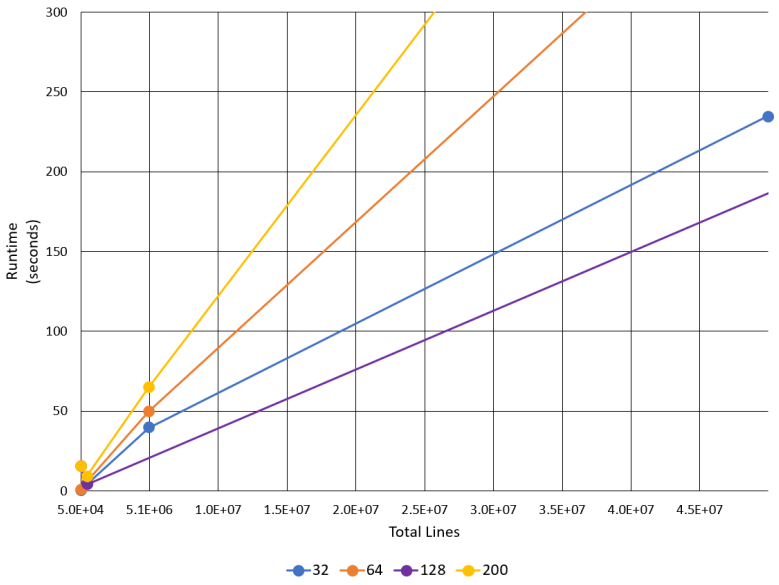


Figure 4: Execution time versus nodes for total lines from 50,000 to 50,000,000

of the output are shown below:

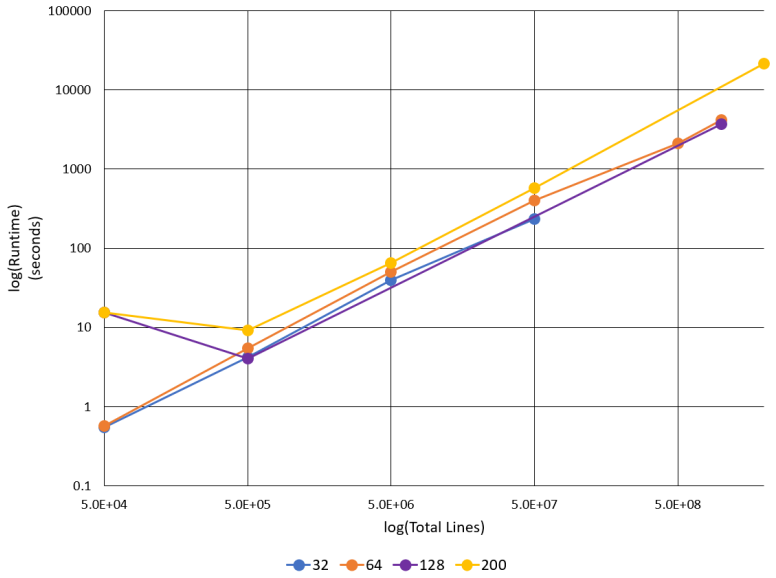


Figure 5: Execution time versus nodes and total lines from 50,000 to 2,000,000,000

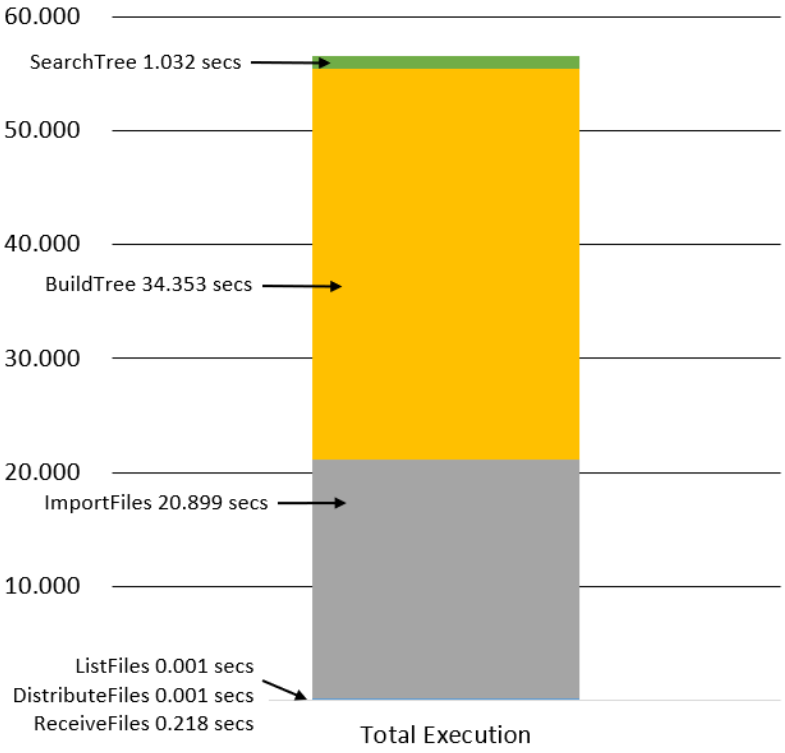


Figure 6: Execution time by function for 5,000,000 total lines and 1,000 search rows

POINTS FOUND:

X	Y	Z	0.01	0.05	0.10
0.581959	0.721012	0.969341	917205	55498018	92120227
0.894312	0.362959	0.447526	13375	1634169	12135389
0.801765	-0.037433	0.616920	4026	490535	3837163
0.685972	0.346683	0.388596	29251	3420777	22785611
0.716828	0.953059	0.275552	3771	462728	3675457
0.113410	0.650905	0.795308	797814	53199022	78642100
0.125404	0.282296	0.536048	1473	187628	1556214
0.557424	0.471667	-0.075745	1559	190467	1523646
0.557737	0.697675	0.934643	819217	52896281	92109359
0.353073	0.840097	-0.000039	1631	195440	1564030
0.365097	0.064303	0.803032	13811	1693414	12855261
0.917744	0.511031	1.053122	180759	18109472	67214583
0.611690	-0.057995	0.263766	1519	186829	1523365

The program was tested using

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 challenges which we struggled with as well as our successes and possible future work.

Challenges:

- `parallelSort` conversions
- memory leaks
- `malloc` when you should `realloc`
- 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

5.1 Recommendations

The following recommendations could improve the course when offered next:

- Assign a job queue to each team with dedicated nodes for each to eliminate resource conflicts between teams.
-

5.2 Links

```
* k-d tree repo
  https://github.com/jjlay/COMS7900kdTree
```

* Production source
  https://github.com/jjlay/COMS7900kdTree/tree/master/code/kdTree/parallelApproved

* Paper
  https://github.com/jjlay/COMS7900kdTree/tree/master/paper

* Presentation
  https://github.com/jjlay/COMS7900kdTree/tree/master/presentation

* ParallelSort
  https://github.com/jjlay/COMS7900kdTree/tree/master/code/parallelSort