

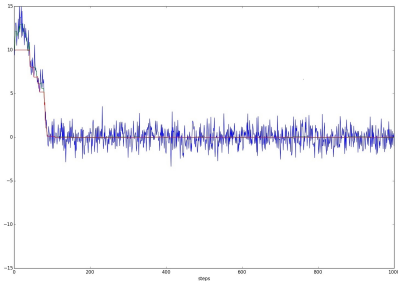
Parallel Orthogonal Recursive Bisection (ORB)

Team Metropolis:

**James Farzi, JJ Lay,
Graham West**

April 19, 2019

COMS 7900, Capstone



Outline

1 Introduction

2 Implementation

- main
- Importing Data
- Tree Structure
 - Building the tree
- Parallel sorting
 - Changes
 - adaptBins
- Searching the tree

3 Validation

- MATLAB visualizations
- Other tests

4 Results

5 Conclusions

- Challenges
- Successes
- Future Work

Introduction

Introduction

Expansion upon previous parallel sorting project

Objectives:

- Given a large set of data
- Develop parallel orthogonal recursive bisection (ORB) algorithm
- Utilize a k -d tree to organize data
- Maximize use of MPI using multiple nodes
- Require both serial/parallel build/search operations
- Search the k -d data with a list of points and 3 radii

Introduction

Workflow:

- Prototyping: implementation based on Graham's MATLAB code
- Extreme coding is FUN...and powerful
- Used C++ w/ C MPI calls
- Using Git effectively:
 - Master and sub branches
 - Reduced merge conflicts
- Execution:
 - `qlogin`
 - `qsub`
- Debugging:
 - `valgrind`
 - `gdb`
- Output sorting:
 - `sleep(myRank)`
 - Prepend a numeric key on `cout`'s

Implementation

Our `main` was quite simple due to our organization of the project into many levels of functions

We also were able to use much of the basic initialization and data importing functions from the previous project

Algorithm 1: `main(...)`

- 1: Initialize MPI
 - 2: Set number of files, lines per file to read
 - 3: import the *data*
 - 4: Initialize *tree*
 - 5: `buildTree(data, tree, comm, ...)`
 - 6: Search the tree with `search501(tree, ...)`
 - 7: Finalize MPI
-

Importing Data

Importing the data:

`listFiles(...)`

- Fetches a list of data filenames using OS calls (random order)

`distributeFiles(...), receiveFiles(...)`

- Isend/Recv the list of filenames
- Round robin distribution of files

`importFiles(...)`

- Reads the received filenames
- Read a set `nFiles` and `nLinesPerFile`
- Returns a 1D array of length $4 \times nFiles \times nLinesPerFile$
- $nFiles \geq nNodes$

`CalculateIndex(...)`

- Calculates the starting index of the first row

listFiles

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

- Fetches a list of data filenames using opendir from path
- Files are returned in “random” order
- Returns list as a vector of string
- Maximum number of files returned is numFiles

distributeFiles

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

- Asynchronously transmits filenames in `files` to nodes
- Used `MPI_Isend` since it sends to itself
- Round robin distribution of files
- Transmits the string "DONE!" to all nodes at end

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

- Asynchronously receives filenames from rank 0
- Used MPI_Irecv and MPI_Wait

importFiles

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

- Read a set nFiles and nLinesPerFile
- Returns a 1D array of length $4 \times \text{nFiles} \times \text{nLinesPerFile}$
- $\text{nFiles} \geq \text{nNodes}$

```
float CalculateIndex(string filename)
```

- Calculates the starting index (I_0) of the first row of the given file
- If x is the five digit integer in the filename and R_{\max} is the maximum number of rows in a file, then
$$I_0 = (x - 1) * R_{\max}$$
- Returns a float to avoid using a separate data type for the index and allows using a single 1D array for the four fields

Tree Structure

Old tree struct:

- Contained extra debugging fields
- Contained completely unused fields
- Originally used doubles
- tree naming

Tree Structure

New tree struct:

```
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  
}
```

definitions.h:

A header file containing numerous preprocessor identifiers (`#define`) to improve code readability and reduce the number of constant values:

- Array indexing
 - `#define _X_ 1` →
 `var = data[_X_]`
 - `#define mpi_Max_Filename 200` →
 `auto name = new char[mpi_Max_Filename]`
- MPI tags
 - `#define mpi_Tag_File 30` →
 `MPI_Send(name, sz, MPI_CHAR, Rank0, mpi_Tag_File, ...)`
- Count limits
 - `#define abortCount 5000` →
 `while (i < abortCount)`

Building the tree

To build the tree, we use several functions which perform different aspects/sections of the task

Functions:

- `buildTree`
- `buildTree_serial`
- `buildTree_parallel`
- `getSortDim`

Building the tree

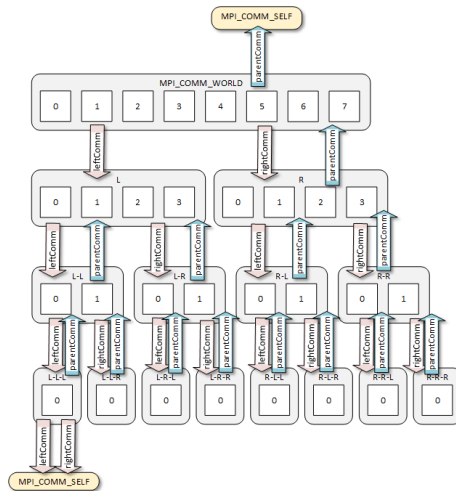


Figure 1: Example of parallel variables using eight nodes

Building the tree

`buildTree` checks the number of compute nodes in the current communicator and determines whether to call the parallel or serial versions of the code

Algorithm 2: `buildTree(data, tree, comm, ...)`

```
1:  $q = \text{Size of current communicator}$ 
2: if  $q > 1$  then
3:   buildTree_parallel(data, tree, comm, ...)
4: else
5:   buildTree_serial(data, tree, ...)
6: end if
```

Building the tree

`buildTree_parallel` performs ORB using a multiple compute nodes

Algorithm 3: `buildTree_parallel(data, tree, comm, ...)`

```
1: Call getSortDim(...): calculates  $x, y, z$  mins, maxs, ranges, partition center, and
   returns sortDim
2: Sort data over sortDim using parallelSort(data, sortDim, comm, ...)
3: if myRank < numNodes/2 then
4:   Create tree.L, commL
5:   buildTree_parallel(data, tree.L, comm, ...)
6: else
7:   Create tree.R, commR
8:   buildTree_parallel(data, tree.R, comm, ...)
9: end if
```

It is assumed that $tree.n > 1$ will never occur in `build/tree_parallel` since we usually deal with large amounts of data

Building the tree

`buildTree_serial` performs ORB using a single compute node

Algorithm 4: `buildTree_serial(data, tree, ...)`

```
1: if tree.n > 1 then
2:   Calculate x, y, z mins, maxs, ranges, and partition center
3:   Sort data over sortDim =  $\text{argmax}(x, y, z \text{ ranges})$ 
4:   Split data: dataL, dataR
5:   if  $|dataL| > 0$  then
6:     Create tree.L
7:     buildTree_serial( dataL, tree.L, ... )
8:   end if
9:   if  $|dataR| > 0$  then
10:    Create tree.R
11:    buildTree_serial( dataR, tree.R, ... )
12:   end if
13: else
14:   Store data (a single point)
15: end if
```

Building the tree

`getSortDim` finds the longest axis and stores several key tree fields

Algorithm 5: `getSortDim(data, tree, comm, ...)`

- 1: Each process gets its local x, y, z min and max
 - 2: Rank 0 receives these, determines the global x, y, z min and max, determines the `sortDim`, and Bcast's all of these values back to the other nodes
 - 3: The global mins/maxs, partition center, and partition radius are stored in *tree*
 - 4: return *sortDim*
-

Parallel sorting

We had to make several significant alterations to our `parallelSort` program in order to integrate it into our KD tree project

Changes:

- Make rank 0 do work
- Use specified communicator
- Conversion to function
- better `adaptBins`

Making rank 0 do work:

- Initially, rank 0 was just a master node which coordinated the other worker nodes
- This technique is very inefficient for parallel ORB since it requires us to switch to serial mode sooner
- The solution involved 1) cleverly altering a large number of if statements in the code and 2) changing how certain types of sends/recvs were handled

Using a specified communicator:

- Initially, `parallelSort` and all of its associated functions used `MPI_COMM_WORLD` (hard-coded)
- To use a specified communicator *comm*, it must be passed as an argument into any function that uses it
- This required a simple but tedious process of editing

Building the tree

Here is how `parallelSort` is structured now that it is a function

Algorithm 6: `parallelSort(data, rows, myRank, sortDim, comm, ...)`

- 1: Locally sort *data* on each compute node using a qsort
 - 2: Determine the global min/max of the *sortDim*
 - 3: Create linearly spaced bin edges over range on rank 0 and Bcast
 - 4: Bin the *data* on each compute node and accumulate on rank 0
 - 5: Calculate *uniformity*
 - 6: **while** *uniformity* < *threshold* && *iterations* < *M* **do**
 - 7: Adapt the bin edges on rank 0 and Bcast
 - 8: Bin the *data* on each compute node and accumulate on rank 0
 - 9: Calculate *uniformity*
 - 10: **end while**
 - 11: Swap *data* between compute nodes and do data cleanup
-

Parallel sorting

We also wished to modify our original adaptBins function

Old adaptBins:

- Local method
- Based on the normalized gradient of the bin counts
- Scaled so that bin edges remain properly ordered
- Scale decreases over time to avoid oscillations
- **Pros:** able to handle nonlinearities in distribution, good at fine-tuning
- **Cons:** edges from dense regions are slow to converge, slower with more nodes

$$\begin{aligned}\Delta C &= 2.0(C_{i+1}^m - C_i^m)/(C_{i+1}^m + C_i^m) \\ \Delta E &= E_{i+1}^m - E_i^m \\ S(m) &= 1 - (1 - 0.1)(1 - \exp(-0.03m)) \\ E_i^{m+1} &= E_i^m + 0.475(S(m)\Delta C\Delta E)\end{aligned}\tag{1}$$

Parallel sorting

We also wished to modify our original adaptBins function

New adaptBins:

- Global method
- Based on the integrated, linearly interpolated, cumulative distribution
- Bin edges placed where linear interpolation would assume uniformity
- **Pros:** fast initial convergence in approximately linear regions, same speed with more nodes
- **Cons:** can oscillate near dense regions

$$\hat{C}(x) = \hat{C}(E_{i'}^m) + C_{i'}^m \frac{x - E_{i'}^m}{E_{i'+1}^m - E_{i'}^m} = (i+1) \frac{D}{N} \quad (2)$$

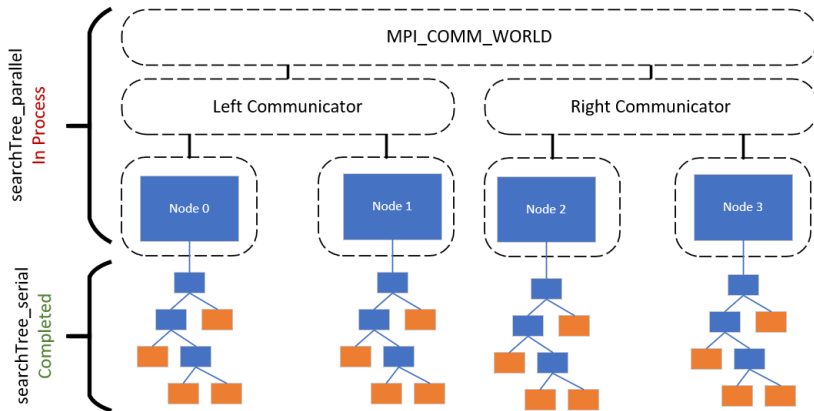
$$E_i^{m+1} = E_{i'}^m + \left((i+1) \frac{D}{N} - C(E_{i'}^m) \right) (E_{i'+1}^m - E_{i'}^m) / C_{i'}^m \quad (3)$$

Solution:

- Alternate between the old and new schemes on even and odd iterations
- MATLAB demos

Searching the tree

Each node calls `search501`, reads the 501-st data file, and loops through the points contained within (as well as the three given radii). For each point and radii, it calls `searchTree_serial`.



Searching the tree

`searchTree_serial` returns the number of points from the local tree within a given radius about a given point.

Algorithm 7: `searchTree_serial(tree, rad, point)`

```
1: found = 0
2:  $d = \sqrt{\sum_{i=1}^3 (point[i] - tree.c[i])^2}$ 
3: if  $d \leq rad + tree.rad$  then
4:   if  $tree.L = NULL \ \&\& \ tree.R = NULL$  then
5:     return 1
6:   else
7:     if  $tree.L \neq NULL$  then
8:       found += searchTree_serial(tree.L, rad, point)
9:     end if
10:    if  $tree.R \neq NULL$  then
11:      found += searchTree_serial(tree.R, rad, point)
12:    end if
13:  end if
14: end if
```

`searchTree_parallel` was in development:

- Tasks completed
 - The nodes successfully divided into groups of communicators
 - The nodes organized data correctly
 - Local trees were built
- Tasks remaining
 - Rank 0 only reader of 501
 - Push the id of Rank 0 for the left and right communicators up
 - Implement MPI calls between parent and children communicators

Validation

MATLAB Demos:

- 2D animation of MATLAB prototype
- 3D visualization at the end of the parallel phase of the C++ implementation

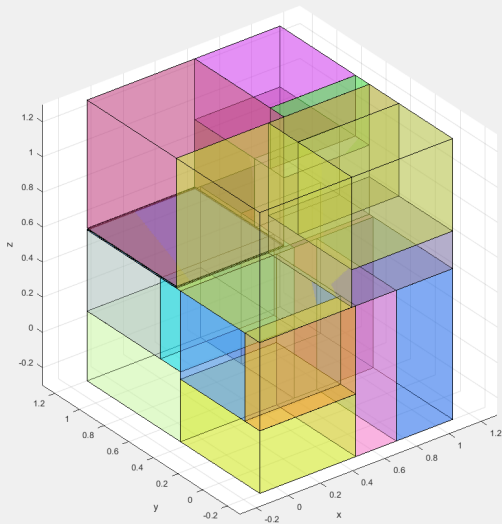


Figure 2: Example of k -d tree partitions

Other validation methods:

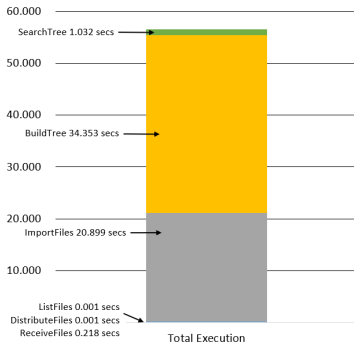
- Set search sphere center on a data point, use tiny radius, find 1 point
- Set an arbitrary search sphere point, increase radius, points found increases monotonically
- Consistent results for different numbers of nodes
- `dumpTree` writes each compute node's tree to a file for analysis

Results

Timing Data

By function:

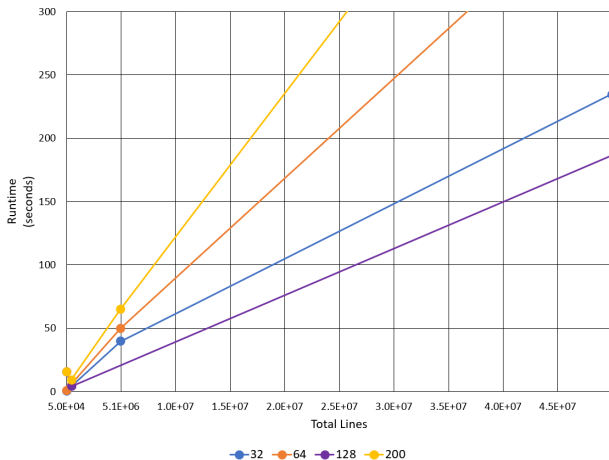
- Test data was 5,000,000 total data rows and 1,000 search rows on 4 cores
- I/O was the primary bottleneck: disk for `importFiles` and network for `buildTree`



Timing Data

Performance:

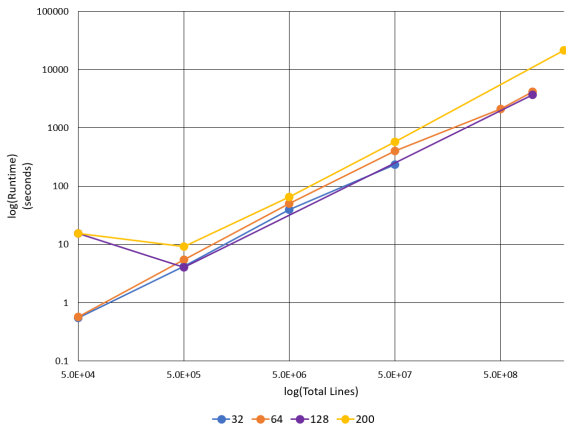
- For fewer than 45,000,000 data rows, 128 nodes performed best



Timing Data

Performance:

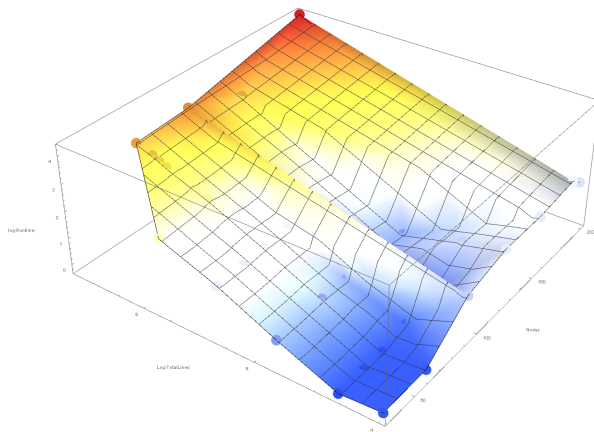
- Overall, 32 cores performed equivalent to 128 cores
- 200 cores was the worst performer



Timing Data

Scalability:

- Overall, 32 cores performed equivalently to 128 cores
- 200 cores was the worst performer



Search Results

Output:

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

Conclusions

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

Challenges

Obnoxious personality quirks and coding habits that were not ship-shape:

- Graham:
 - Occasionally becomes side tracked addressing small things
 - So many windows open! How do you find anything?
 - Always has to try it the wrong way before doing what James suggested 10 minutes earlier
 - Makes an “improvement”, 15 minutes later reverts the ‘improvement’ in order to improve - because there *IS* a correct way to organize sand
- James:
 - Who needs whitespace?
 - Pointers! Pointers everywhere!
 - Memory leak? Just run it on more nodes!
 - Hey! Who put `sleep(myRank)` here?
- JJ:
 - Insistent usage of Visual Studio replaces neatly arranged tabs with inconsistent spaces (also doesn't update Makefile)
 - Debug cout hoarder
 - Naming conventions? Who needs those? (What's a camel case?)

Successes:

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

Helpful personality quirks and proper coding habits:

- Graham:
 - I know math (show me)
 - Very detailed oriented (my weakness has become my strength)
 - VIM, TeX wizard
 - How does this guy know the differences in `exampleblock`, `alertblock`, and `block`?
- James:
 - Engineering mindset provides more efficient solutions to problems
 - Always seems to know where the bugs are
 - Able to keep Graham from organizing all the sand
- JJ:
 - C++, Git, Excel, MPI guru
 - Literally a human library for terminal commands

Future work:

- cloud computing
- use of coding techniques for personal research