

# Parallel Orthogonal Recursive Bisection

Team Metropolis:  
James Farzi, JJ Lay, Graham West

April 10, 2019

# Outline

## 1 Introduction

## 2 Implementation

- main
- KD tree
  - Building the tree
  - Searching the tree
- Parallel sorting
  - Changes
  - adaptBins

## 3 Validation

- MATLAB visualisations
- Other tests

## 4 Results

## 5 Conclusions

- Challenges
- Successes
- Future Work

# Introduction

# Introduction

Expansion upon previous parallel sorting project

## Objectives:

- Given 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

## SUBLISTSSSSSS

### Workflow:

- Using Git effectively (multiple branches, few merge conflicts)
- Extreme coding if FUN...and powerful
- Prototyping: we each wrote our own, implementation based on Graham's MATLAB code

# Implementation

# Building the tree

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
-

# 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

`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(...)`

---

```
1:  $q$  = Size of current communicator
2: if  $q > 1$  then
3:   buildTree_parallel(...)
4: else
5:   buildTree_serial(...)
6: end if
```

---

# Building the tree

`buildTree_serial` performs ORB using a single compute node

---

**Algorithm 3:** `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

`buildTree_parallel` performs ORB using a multiple compute nodes

---

**Algorithm 4:** `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

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*
-

# Searching the tree

`searchTree_serial` returns the number of points within a given radius about a given point

---

**Algorithm 6:** `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
```

---

# Searching the tree

search501 reads the 501-st data file and loops through the points contained within (as well as the three given radii), calling `searchTree_serial` for each

---

**Algorithm 7:** `search501(tree, path, ...)`

---

1:

---

# 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 8:** `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
- **New:** Scaled to decrease of time to avoid oscillations
- **Pros:** able to handle nonlinearities in distribution, good at fine-tuning
- **Cons:** edges from 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

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

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

# Parallel sorting

Since both methods' pros and cons are disjoint, alternating between them gives a method which can outperform either individually

## Demos:

- thin dense distribution at boundary,  $nBins = 2, 3, 10$
- wide distribution in center,  $nBins = 2, 3, 10$
-

# Validation

## MATLAB Demos:

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

## 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
- dumpTree writes each compute node's tree to a file for analysis



# Results

# Conclusions

# Challenges

## 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

## Future work:

- cloud computing
- use of coding techniques for personal research