Parallel Orthogonal Recursive Bisection

Team Metropolis:
James Farzi, JJ Lay, Graham West

April 10, 2019

Outline

Introduction

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: $\texttt{main}(\cdots)$

1: Initialize MPI
2: Set number of files, lines per file to read
3: import the $data$
4: Initialize $tree$
5: $\texttt{buildTree}(data,\ tree,\ comm,\ \cdots)$
6: Search the tree with $\texttt{search501}(\ tree,\ \cdots)$
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($\cdots$)

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

Building the tree

`buildTree_serial` performs ORB using a single compute node

Algorithm 3: `buildTree_serial`($data$, $tree$, $\cdots$)

1: **if** $tree.n > 1$ **then**
2:   Calculate x, y, z mins, maxs, ranges, and partition center
3:   Sort $data$ over $sortDim = \arg\max(x, y, z \text{ ranges})$
4:   Split $data$: $dataL, dataR$
5:   **if** $|dataL| > 0$ **then**
6:    Create $tree.L$
7:    `buildTree_serial`($dataL$, $tree.L$, $\cdots$)
8:   **end if**
9:   **if** $|dataR| > 0$ **then**
10:    Create $tree.R$
11:    `buildTree_serial`($dataR$, $tree.R$, $\cdots$)
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`($data$, $tree$, $comm$, $\cdots$)

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

1: Each process gets it 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** $tree.L \mathrel{!=} NULL$ **then**
8:          $found \mathrel{+}= $ searchTree_serial($tree.L$, rad, $point$)
9:       **end if**
10:     **if** $tree.R \mathrel{!=} NULL$ **then**
11:        $found \mathrel{+}= $ 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($tree$, $path$, $\cdots$)

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`

Parallel sorting

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

Parallel sorting

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`($data$, $rows$, $myRank$, $sortDim$, $comm$, $\cdots$)

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
- **Cons:** edges from from dense regions are slow to converge

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

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
- **Cons:** oscillates near dense regions

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

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

Parallel sorting

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

Demo:

Old:

- Works:
- Fails:

New:

- Works:
- Fails:

Validation

Validation

MATLAB Demos:

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

Validation

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