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 visualitations
   - Other tests
4. Results
5. Conclusions

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: `buildTree`$(\cdots)$

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

if $tree.n > 1$ **then**
  Calculate x, y, z mins, maxs, ranges, and partition center
  Sort $data$ over $sortDim = \text{argmax}(x, y, z \text{ ranges})$
  Split $data$: $dataL, dataR$
  if $|dataL| > 0$ **then**
    Create $tree.L$
    `buildTree_serial`($dataL$, $tree.L, \cdots$)
  end if
  if $|dataR| > 0$ **then**
    Create $tree.R$
    `buildTree_serial`($dataR$, $tree.R, \cdots$)
  end if
else
  Store $data$ (a single point)
end if

Building the tree

`buildTree_parallel` performs ORB using a multiple compute nodes

Algorithm 4: `buildTree_parallel`($data$, $tree$, $comm$, $\cdots$)

```
Call getSortDim(···): calculates x, y, z mins, maxs, ranges, partition center, and
returns sortDim
Sort data over sortDim using parallelSort(data, sortDim, comm, ··· )
if myRank < numNodes/2 then
    Create tree.L, commL
    buildTree_parallel( data, tree.L, comm, ··· )
else
    Create tree.R, commR
    buildTree_parallel( data, tree.R, comm, ··· )
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`$(tree, rad, point)$

$found = 0$
$d = \sqrt{\sum_{i=1}^{3}(point[i] - tree.c[i])^2}$
if $d \leq rad + tree.rad$ **then**
    if $tree.L = NULL$ && $tree.R = NULL$ **then**
        return 1
    else
        if $tree.L$!= $NULL$ **then**
            $found$ += `searchTree_serial`$(tree.L, rad, point)$
        end if
        if $tree.R$!= $NULL$ **then**
            $found$ += `searchTree_serial`$(tree.R, rad, point)$
        end if
    end if
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
- 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

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:

rank 0 multiple communicators
old new alternating

Validation

Validation

MATLAB Demos:

- 2D
- 3D

tiny/huge radii multiple nodes

Results

Conclusions