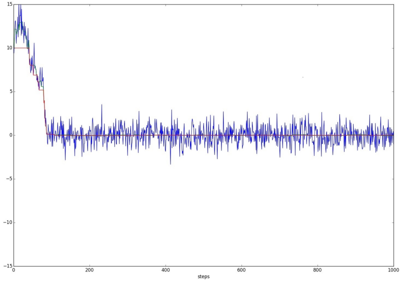
Parallel Orthogonal Recursive Bisection (ORB)

Team Metropolis:
James Farzi, JJ Lay, Graham West

April 19, 2019

COMS 7900, Capstone



Outline

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
- Requires both serial/parallel build/search operations
- Search for nearest neighbors in 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

`main`

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`$(\cdots)$

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

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

listFiles($\cdots$)

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

distributeFiles

`distributeFiles(···)`, `receiveFiles(···)`

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

receiveFiles

distributeFiles($\cdots$), receiveFiles($\cdots$)

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

`importFiles`

`importFiles`$(\cdots)$

- 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

`CalculateIndex(···)`

- Calculates the starting index of the first row

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
}
```

C Constants

`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

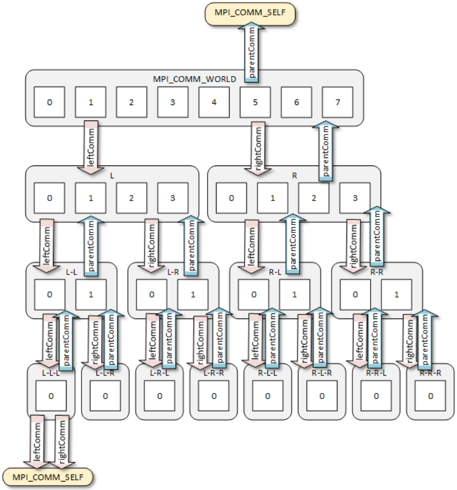


Figure 1: Example of parallel variables using eight nodes

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`$(data, tree, comm, \cdots)$

```
1: q = 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 multiple compute nodes

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

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

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

`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 $tree.L \mathrel{!}= NULL$ **then**
      $found \mathrel{+}=$ `searchTree_serial`$(tree.L, rad, point)$
    end if
    if $tree.R \mathrel{!}= NULL$ **then**
      $found \mathrel{+}=$ `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
- 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 the to switch to serial mode to occur earlier in the tree
- The solution involved 1) cleverly altering a large number of if statements in the code, 2) changing many loops to begin at 0 rather than 1, and 3) 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`($data$, $rows$, $myRank$, $sortDim$, $comm, \cdots$)

```
Locally sort data on each compute node using a qsort
Determine the global min/max of the sortDim
Create linearly spaced bin edges over range on rank 0 and Bcast
Bin the data on each compute node and accumulate on rank 0
Calculate uniformity
while uniformity < threshold && iterations < M do
    Adapt the bin edges on rank 0 and Bcast
    Bin the data on each compute node and accumulate on rank 0
    Calculate uniformity
end while
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 from dense regions are slow to converge, slower with more nodes

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

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

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

$$E^{m+1}_i = E^m_{i'} + \left((i+1)\frac{D}{N} - C(E^m_{i'})\right)(E^m_{i'+1} - E^m_{i'})/C^m_{i'} \tag{3}$$

Parallel sorting

Solution:

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

Validation

Validation

MATLAB Demos:

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

Validation

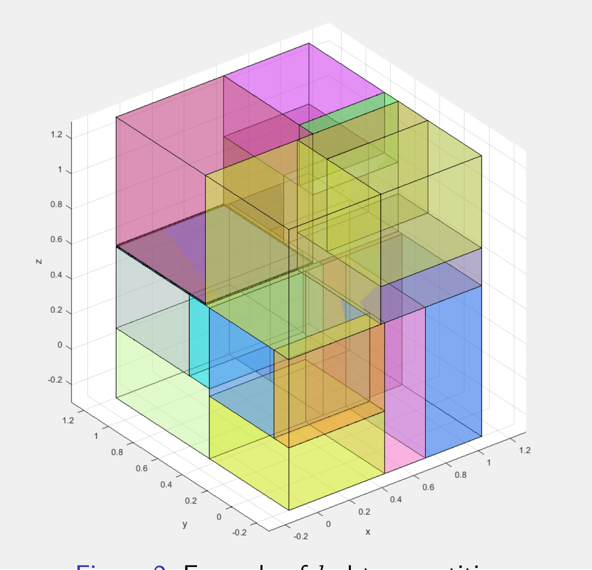


Figure 2: Example of k-d tree partitions

Validation

Other validation methods:

- Set radius very large, find all points
- 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
- Vary number of compute nodes (altering tree structure), find consistent number of points
- `dumpTree` writes each compute node's tree to a file for analysis

Results

Conclusions

Challenges

Challenges:

- memory management (leaks, limited space, Signal 9)
- array out of bounds issues
- multiple communicators (comm)
- no planning for function arguments and return values (constant editing of h-files)
- testing was difficult due to cluster overloading and hardware errors
- debug print statement clutter
- inconsistent usage pointer-to-pointer calls for *data[] and *rows (due to `swapArrayParts`)
- `malloc` when you should `realloc`
- `parallelSort` conversions
- `adaptBins` convergence problems

Challenges

Annoying personality quirks and idiotic coding habits:

- Graham:
  - Obsessive fixation on the most minor of convention details (can't see the forest for the trees)
  - 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
- James:
  - Who needs whitespace?
  - Pointers! Pointers everywhere!
  - Memory leak? Just run it on more nodes!
- JJ:
  - Insistent usage of Visual Studio replaces neatly arranged tabs with ugly, inconsistent spaces
  - SO MUCH debug/timing cout clutter
  - Naming conventions? Who needs those? (What's a camel case?)

Successes

Successes:

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

Successes

Helpful personality quirks and proper coding habits:

- Graham:
  - I can do math
  - Obsessive attention to detail pays off when writing the paper/presentation
- James:
  - Engineering mindset provides more efficient solutions to problems
- JJ:
  - C++/Git guru

Future Work

Future work:

- cloud computing
- use of coding techniques for personal research