Parallel Orthogonal Recursive Bisection

Team Metropolis:
Jamshid 'James' Farzidayeri, JJ Lay, and Graham West

COMS 7900, Capstone

Abstract

In our first project, we implemented a parallel sorting algorithm which utilized a local gradient-type optimization search to equalize the amount of data across different compute nodes in order to achieve maximum efficiency. In this project, we applied this algorithm to the problem of parallel orthogonal recursive bisection (ORB), i.e., the construction of k-d trees. In order to do this, we had to heavily modify the sorting algorithm in several ways, including 1) turning it into a callable function, 2) letting the rank 0 head node perform work while still managing the tasks, 3) incorporating the use of different MPI communicators, and 4) altering the adaptive binning technique for better convergence.

In this paper, we will discuss how our k-d tree algorithm works, how we solved the various issues plaguing parallel sort (mentioned above), and how we tested and validated our work. We conclude with a discussion of the major difficulties in completing this project and how these difficulties could be minimized in the future.

Contents

1 Introduction

```
walk = RandomWalk(nStep, initPos, stdDev)
```

STATEMENT OF THE PROBLEM: talk about orthogonal recursive bisection ORB, needed to search all the data with 3 radii and 501 file centers, return table of counts

1.1 Workflow

How we went about the project...

new GitHub repo

EXTREME CODING FTW: catch errors quickly, no merge conflicts, everyone writes/knows all the code

used a parallelApproved dir in repo to minimize merge conflicts

went to WPS instead of SCI due to better work environment and longer work time

We all wrote prototypes in several languages before attempting the full C++ parallel version. We chose to follow the format Graham used in his MATLAB implementation.

1.2 Variables and conventions

Counts:

- N: number of nodes
- M: max number of allowed time steps
- L: number of lines to read per file
- L_w: number of lines on the wth worker
- D: total number of lines/data points

Indices:

- $m = 0, \cdots, M$ is the time step of the bin adaptation scheme (likely less than M)
- $n = 0, \cdots, W$ spans the nodes
- $i = 0, \cdots, N$ spans the bin edges/indices
- $j = 0, \cdots, N - 1$ spans the bin counts (this will occasionally subscript binI/E as well)
- $\ell_w = 0, \cdots, L_n - 1$ spans the lines on the n-th node
- $k = 0, \cdots, 3$ is the data column being sorted

Variables:

- $\text{data}^n_{4\ell+k}$: data point on ℓth line and kth column on the n-th node
- E^m_j: bin edges (0 indexed) at time step m
- $I^{n,m}_j$: bin indices on node n
- $C^{n,m}_j$: bin counts on node n at time step m
- C^m_j: total bin counts on head node (sum of node C's) at time step m

2 Implementation

Here we discuss our implementation of the code

we used C++ with C MPI calls, mpirun, qsub/qlogin, valgrind

2.1 `main`

import: gets the list of files and imports their data into an array

build: we build the tree using ORB to partition the data

search: import the 501-st data file and perform a search on each, summing the total number of points found over several radii

2.2 K-d tree

Used a Tree struct...

2.2.1 Building the tree

2.2.1.1 `buildTree` This function gets the number of compute nodes q available in the current communicator and determines which function to run. If $q > 1$, then we can still do a parallel sort with 2 compute nodes, so `buildTree_parallel` is entered. If $q = 1$, then `buildTree_serial` is entered.

2.2.1.2 `buildTree_serial` This was the first function written after our initial prototyping phase was completed. It essentially performs a serial version of ORB which can be executed on a single compute node.

Upon completion, `buildTree_serial` calls itself instead of `buildTree` since we still have $q = 1$.

2.2.1.3 `buildTree_parallel` This function performs essentially the same tasks as `buildTree_serial`, but utilizing multiple nodes for speedup. Specifically, it takes advantage of `parallelSort` in order to speed up the partitioning of the data along the longest axis (determine by `getSortDim`, discussed below). After the data has been sorted, the data is split by placing lower half (w.r.t. the sorted data values) of the compute nodes into a left communicator and the upper half into a right communicator. Each half then calls `buildTree`.

2.2.1.4 `getSortDim` This function is used by `buildTree_parallel` in order to determine which is the longest axis, i.e., the sort dimension. In addition to this, it also fills in the struct for the current tree node with the global min/max in all three dimensions. To do this, each node gets its own min/max for each axis and sends them to rank 0 (w.r.t. the current communicator). Rank 0 then determines the global min/max and them MPI_Bcast's those values back to the other ranks. This allows each communicator to sort their data independently along different axes. The center of the bounding box defined by these values min's/max's is also stored in the tree struct.

2.2.2 Searching the tree

2.2.2.1 `searchTree_serial` This function is called by all ranks and returns an integer which equals the number of points found by the rank that called it. It takes as arguments the point and radius (which specifies a search sphere) and the root of the tree created by `buildTree`. To perform the search, a check is done to determine of the search sphere intersects the bounding sphere of each data partition:

$$test \tag{1}$$

If this check returns false, then there are no data points within the search sphere contained in that partition. If true, then another check is performed to check if the partition contains only a single point (if so, it increments the found count); otherwise the search branches and calls itself, creating a recursive loop.

ADD PSEUDOCODE

2.2.2.2 `search501`

Each node calls this function which reads the 501-st data file, whose contents are the centers of the search spheres. It then loops through `searchTree_serial` through all of the sphere centers for three different radii (0., 0., 0.). The counts on each node are then stored. After all searches are complete, then an MPI_Reduce is performed by all nodes, thus adding all of the counts into a single array which is exported to a file.

2.3 Altered parallel sorting

We had to adjust our parallel sorting algorithm a great extend to integrate it into the new project.

2.3.1 `parallelSort`

2.3.1.1 Conversion to function

This was easiest aspect, since we merely found all of the required params...

the one difficulty was handling the *data[] thing

2.3.1.2 Making rank 0 do work

Why do this: super inefficient

this required some clever rearrangement of if statements in several sections of the code, ost notably, the adaptive binning section

2.3.1.3 Using different communicators

Since kdtree requires multiple comms, it was necessary that parallel sort (any many functions it contains) was given the current communicator. This was not a terribly difficult modification, but it was quite tedious since there were many functions which required alterations to their header files.

2.3.2 `adaptBins`

Although our original adaptive binning scheme performed well, we wanted something which could do even better since k-d trees require many parallelSort calls. As a review, here is our original method:

$$\begin{aligned}\Delta C &= 2.0(C^m_{i+1} - C^m_i)/(C^m_{i+1} + C^m_i)\\ \Delta E &= E^m_{i+1} - E^m_i\\ E^{m+1}_i &= E^m_i + \alpha \Delta C \Delta E\end{aligned} \tag{2}$$

where C are the bin counts, E are the bin edges, and $\alpha < 0.5$. This method will occasionally devolve into oscillatory behavior and not converge to the correct value. To combat this in the k-d tree project, we added a scale factor S which decreases over time:

$$\begin{aligned}S &= 1 - (1 - 0.1)(1 - \exp(-0.03m))\\ E^{m+1}_i &= E^m_i + \alpha S \Delta C \Delta E\end{aligned} \tag{3}$$

Since this method is local, it converges slowly at bin edges far away from high-density clusters. As such, we attempted to replace this method with a global method which uses linear interpolation to estimate where the bins would be evenly distributed. Define the function $\hat{C}(x)$ as the linear approximation of the cumulative count distribution of the data points (so that it normalizes to the number of data points, not unity). Then,

$$\hat{C}(x) = \hat{C}(E^m_{i'}) + C^m_{i'} \frac{x - E^m_{i'}}{E^m_{i'+1} - E^m_{i'}} = (i+1)\frac{D}{N} \tag{4}$$

where i' is the maximum index such that $\hat{C}(E_{i'}) < (i+1)\frac{D}{N}$ (note that i' and i are distinct integers). The right equality implies that we should solve for the value of x such that it holds. This x value will be the new i-th bin edge, E^{m+1}_i. Therefore,

$$E^{m+1}_i = E^m_{i'} + \left((i+1)\frac{D}{N} - C(E^m_{i'})\right)(E^m_{i'+1} - E^m_{i'})/C^m_{i'} \tag{5}$$

We discovered that this adaptation technique performs very will in initial steps of adaptation, but is prone to oscillations near clusters of points. Now, since each of the two methods perform better and worse in different contexts, are solution was to simply alternate between at each step. This solved all of our convergence issues in testing. (Note that we still use the same binary search-based binning technique and stopping criterion from the previous project.)

3 Validation

3.1 Two MATLAB demos

Should these not be in the paper since animation is a pain without adobe??? Leave it for the presentation where we can just run it?

3.1.1 2D animation of tree building

3.1.2 3D tree partitions from different orientations

In addition to our use of MATLAB for prototyping, we also used it to validate our tree building algorithm.

3.2 Other

able to get same counts for different number of nodes

4 Results

4.1 Timing Testing

4.2 Search Results

5 Conclusion

5.1 Challenges

5.2 Future Work

6 Bibliography