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

How we went about the project...

new GitHub repo

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

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.

1.1 Variables and conventions

Here we provide a helpful list of conventions, notations, and variable names used throughout this paper.

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

2.1 Altered parallel sorting

2.1.1 `parallelSort`

2.1.1.1 Conversion to function

2.1.1.2 Making rank 0 do work

2.1.1.3 Using different communicators

2.1.2 `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:

$$\begin{aligned} \Delta C &= 2.0(C_{i+1}^m - C_i^m)/(C_{i+1}^m + C_i^m) \\ \Delta E &= E_{i+1}^m - E_i^m \\ E_i^{m+1} &= E_i^m + \alpha \Delta C \Delta E \end{aligned} \tag{1}$$

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:

$$\begin{aligned} S &= 1 - (1 - 0.1)(1 - \exp(-0.03m) \\ E_i^{m+1} &= E_i^m + \alpha S \Delta C \Delta E \end{aligned} \tag{2}$$

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,

$$\hat{C}(x) = \hat{C}(E_{i'}^m) + C_{i'}^m \frac{x - E_{i'}^m}{E_{i'+1}^m - E_{i'}^m} = (i+1)\frac{D}{N} \tag{3}$$

where i' is the maximum index such that $\hat{C}(E_{i'}) < (i+1)\dfrac{D}{N}$ (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_i^{m+1}. Therefore,

$$E_i^{m+1} = E_{i'}^m + \left((i+1)\frac{D}{N} - C(E_{i'}^m)\right)(E_{i'+1}^m - E_{i'}^m)/C_{i'}^m \tag{4}$$

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.)

2.2 K-d tree

Used a Tree struct...

2.2.1 Building the tree

General notes

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).

Upon completion, `buildTree_parallel` has spawned 2 new communicators (a left and a right) each of which has half as many compute nodes as the parent communicator. All of the nodes from each side then call `buildTree` so that they can perform the q check.

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, as well as the center of the bounding box defined by these values.

2.2.2 Searching the tree

2.2.2.1 `searchTree_serial`

2.2.2.2 `searchTree_parallel`

3 Testing and Validation

4 Results

5 Conclusion

6 Bibliography