Justin Baraboo, class ID 2
group 1 Homework 2

October 2, 2016

1 Logistics

See my git hub for the code, input, output and cheat sheet file.
https://github.com/Feronis/ParallelAlgorithms/tree/master/Assignment2

2 Parallel Hash Table

I implemented a parallel double hashing table in Spark.

Solution A hash table is all ready insanely parallelizable as performing the hash functions and dealing with collisions are great for task parallelism. Even if different orders of hashing yields different ordering in the hash table (that is if "Justin" and "John" and "Jacob" hash to the same values, you can get "11-Justin, 15-Jacob, 19-John" or "11-Jacob, 15-John,19-Justin"), searching remains the same (as it can look/delete for every key value in parallel).

Insertion simply takes all elements in parallel and inserts them. This is $O(n - CRCW)$ for n elements generally being $O(1)$ on average. Deletion is always $O(1 - CRCW)$ (or $O($ largest sized partitions we can create $).$) We simply look at every key in our hash set in parallel and see if it matches what we want to delete. Note that if 2 people have the same name, it will delete them both, extra identifies can be added easily if wanted. Instead of having a $deletePara(Stringname)$ we could have $deletePara(Stringname, intID)$ to first find people who have the same name, and then look for id underneath it after finding matches.

Hash functions are great at finding a specific element without relation to others. So searching for a specific person by their name makes a lot of sense for this query. This type of data structure would not be good for finding a range of names in alphabetical order and a tree data structure would be better for that.

Note: in my code implementation, I use Java's hashtable as it has functions to easily add a value to a specific key. So while I do use an existing hashtable, the implementation of how things are hashed inserted/deleted are defined and controlled by me.

3 Parallel BST

I started the workings of a parallel bst in Spark

Solution Binary Search Trees are great for data with a usual metric (such as numbers) where you want to make comparisons between them such as range queries. This structure is at the cost of efficency as it takes $O(h)$ that is the height of the tree to find the element in searching. There are many ways to parallelize bsts as well. I decided to split the tree up into N smaller bsts. You can also split your queries up so you do each at the same time. Currently, I can't figure out how to do a $RDD1.forEach(item....RDD2.forEach(item2....))$ as there are serializibility errors which are daunting. As such, I have the tree parallelized (as that's the harder part). I sort the data (as a tuple of favNum and Person so you can retrieve the person as well) parallellized. I then chop it into chunks sequentially $O(n)$ and make the trees $O(logn)$. I then can run queries on top of it.

It's very superficial right now and doesn't include addition and deletion.

4 Apache Solr

I also indexed the data on Apache Solr.

The screenshots for adding the data, queries, and output are in the github. Overall, Solr worked much faster, qualitatively, but wasn't tested on a large data set (and I didn't count the time starting the solr server which was longer than starting a spark context).