Kurt Andersen
CS 615
PA3 Bucket Sort
March 30, 2017

Introduction:

For this project we were assigned to write a sequential bucket sort. This algorithm takes in any number of the same data type. The data is then organized into different buckets. Each bucket represents a range in the data. In this instance, none of the numbers were greater than 1000, and I used 10 buckets, so each bucket had an even distribution of the values in the range, or as even as it could be.

Process:

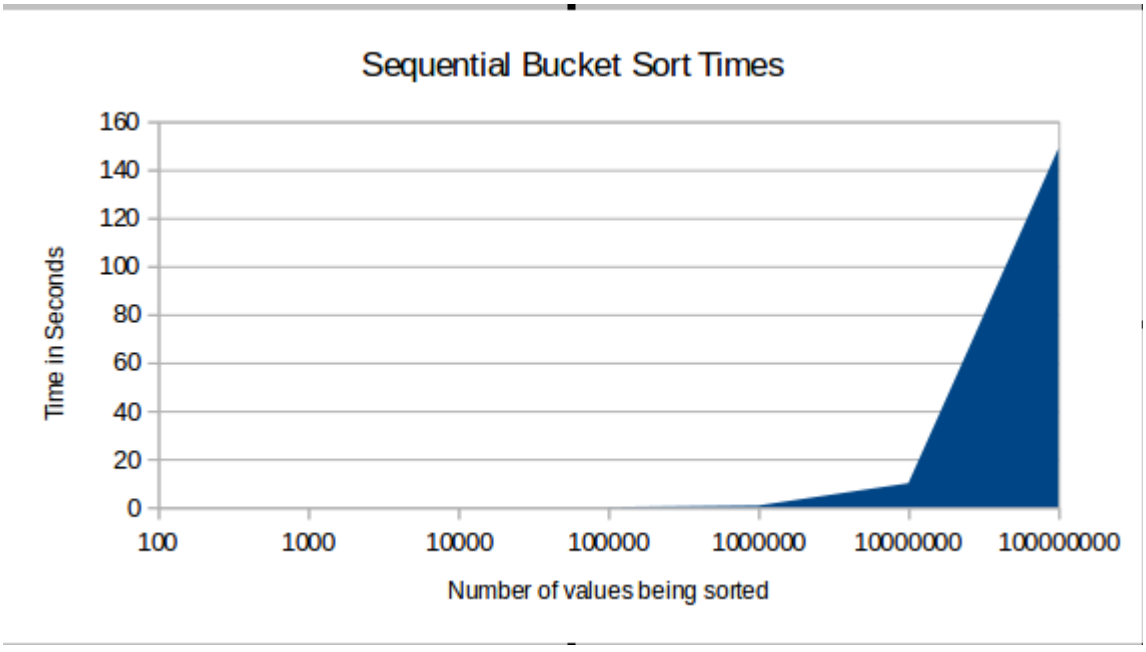
When I started writing this program, I wanted to generate the algorithm myself rather than searching for it on the internet. Once I successfully read in all of the data, I wrote a simple function that found the maximum value in the data set. Once I knew the max value in the data, and how many buckets I needed to organize my data into. I looped through every piece of data in the main array and had a nested loop that would check which bucket the piece of data would be placed into. I had a modular calculation to determine the min and max of each bucket being checked. Once all of the data was organized into the separate buckets, I sorted the data that was stored in each of the buckets. I used standard template libraries list.sort algorithm in order to sort each bucket. The reason I chose this algorithm is because it guarantees an nlogn time complexity. It is important to maintain the same time complexity because we want the same time to organize the same amount of data every time in order to get appropriate readings in our data. If we used a sort such as quick sort, the time it would take for each sort could range anywhere between nlogn to n^2. This would put a massive flaw in the data and it would be unreliable. When the program finishes running, it outputs the time into the following filepath from the main PA3 folder: PA3/bin/outputs. Each of the files were labled with how mana data entries were used.

The only issue I ran into for the sequential portion was when I was calculating the min and max of each bucket. It was not even a programming problem, but just more of an issue on my math logic. I knew what I wanted in my head, i.e. the first bucket would be from 0 to 1/10 of the max value, but

when I had originally started my calculations, my min ended up being 1/2 and my max was 1/3 of the maximum value. So when I ran my tests originally, the data getting sorted was just a hot mess, and was unusable. The only coding issue I ran into was loop logic when I was moving data from the buckets back into the main data array. For my loop limit, I was using the list.size. The issue here was since I was using the list data structure, I had to pop data every time I wanted to get to the next value. This in turn changed the value of the loop limit, and would always stop halfway through the overall iterations. I solved this by simply getting the size and storing it in a static variable when the loop ran.

Results:

For this project I had a sub-exponential growth for my graph. This was entirely expected because the sorting into buckets would take n time complexity, and then the sort would take nlogn time complexity. The overall time complexity of the entire problem would be O(nlogn). When viewing the graph of what nlogn, we can safely say that nlogn = O(n^2) and nlogn = Omega(n).



This graph shows the sub-exponential growth of the algorithm. The x-axis shows how many

integers were read in and then sorted using the sequential bucket sort. While the y-axis is showing the amount of time it took to distribute, sort, then reassemble the data in a sorted fashion using bucket sort. When we start doing the parallel version of this code, the parallel version should start to show some decay in time because this algorithm was not specifically designed to be parallelized.

Conclusion/Future Work:

My future work given this data, will be to conduct the parallel version of this algorithm. I will then interpret and evaluate the data from the parallel version with this sequential version. With my experience from this data that I will be receiving in the future, I will be able to apply this towards other sorting methods as well as other divide and conquer style algorithms.