Ryan Devaney
4/6/17
CS 415
PA03 - Bucket Sort

Description of Assignment

This assignment has us parallelize bucket sort. For this project we first need to write a sequential bucket sort so that we can compare it to a parallelized version of bucket sort. Bucket Sort is a sorting algorithm that runs in O(n+k) time. Bucket sort assumes that there is a well distributed range to work. For my project I used the range of 0 to 10000. Larger and larger amounts of numbers were used in order to compare the running time of sequential bucket sort against a parallelized version of bucket sort.

Part 1 - Sequential Bucket Sort

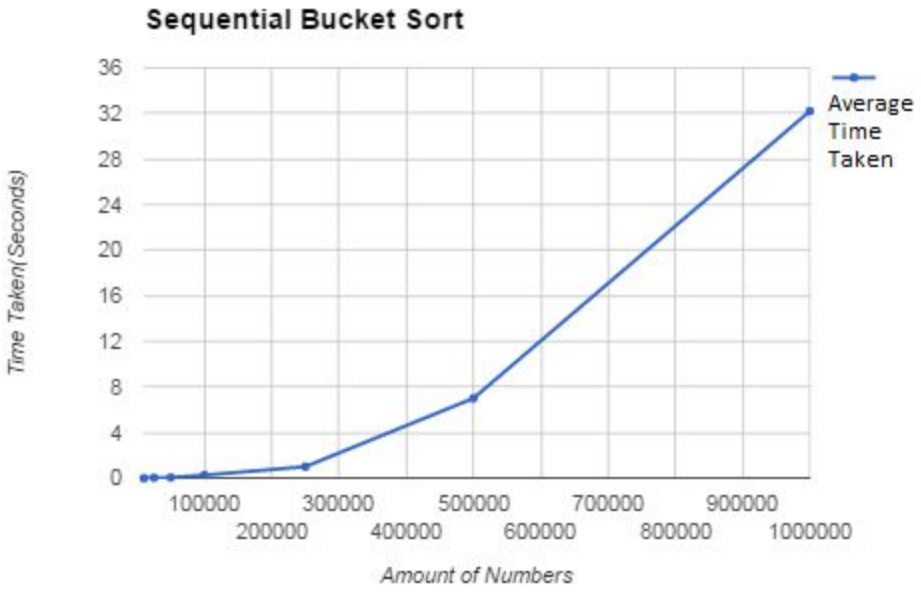
For this part of the project we were to write a sequential bucket sort that would work in the cluster. The first thing to do was write a small program that would easily give a large amount of numbers to a data file. This is a lot better than manually writing out 1 million numbers by hand. The program then has to read in the amount of numbers there are to sort, and then the numbers themselves. After that the numbers are moved to buckets. After that, the buckets are sorted using insertion sort, and concatenated from smallest bucket to largest bucket which results in a sorted list of numbers. Insertion sort was used in order to provide a much better showing of a parallelized version. Insertion sort runs in O(n^2) time and thus when the numbers get large it take a long time to sort. In order to properly time the bucket sort, we take the time from the end of reading in the numbers from the file to when the bucket sort is complete. We do not time IO operations as they would be about the same in sequential and parallel versions of bucket sort and would vary depending on the hardware that you were using.

For gathering times, I used various data sizes ranging from 10000 numbers to 1000000. I tried to go up to 3000000 numbers, but this proved to take to long to get a good enough average.

Numbers	Average Time(s)
10000	0.004463732
25000	0.04596254
50000	0.06521882
100000	0.2736044
250000	1.01341
500000	7.017159
1000000	32.1898

This chart shows the different sizes used and the amount of time taken. These numbers are trending towards an n^2 rate. In addition I ran 3000000 numbers one time and it took

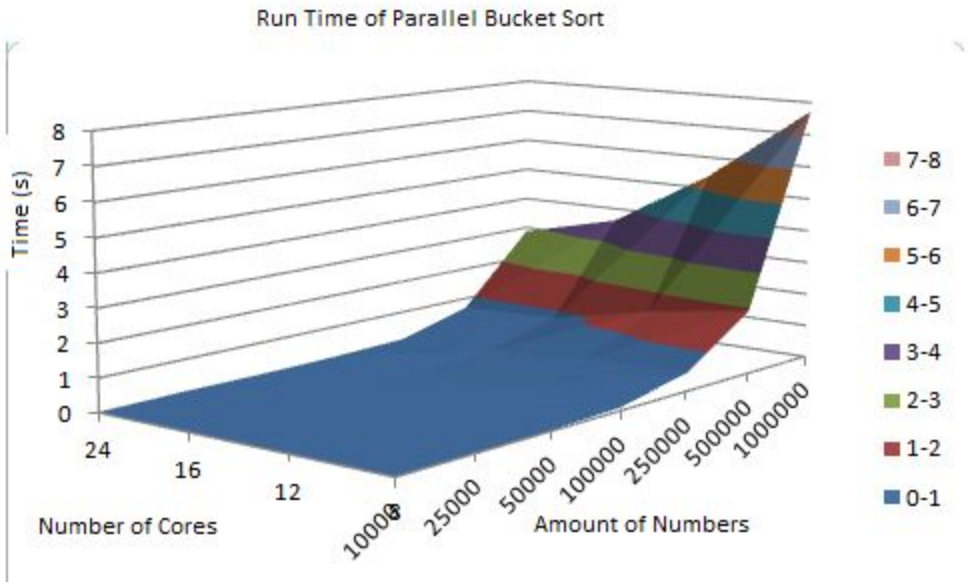
404.918 seconds to run. While this is not an average, it shows just how quickly the time grows as the numbers increase.



This graph shows the time taken plotted against the amount of numbers. It appears to have exponential growth which would make sense since insertion sort was used.

Part 2 - Parallel Bucket Sort

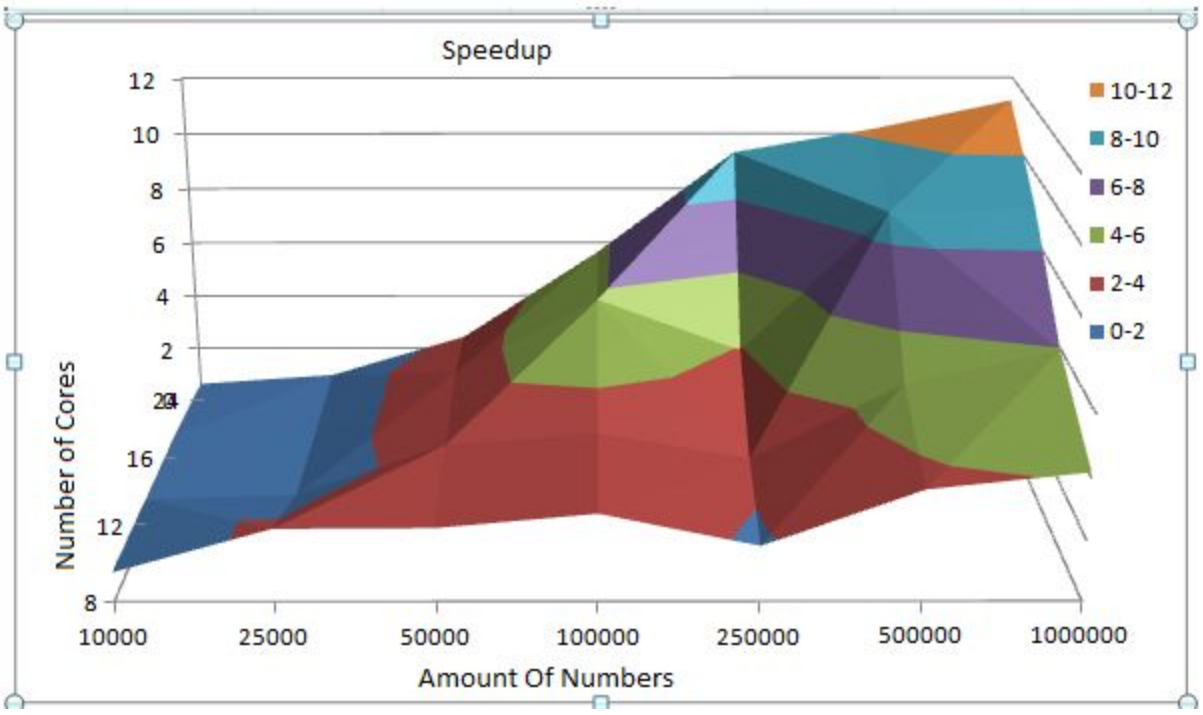
The second part of this project was to parallelize bucket sort using a variety of numbers and a variety of processors. In order to properly compare it to sequential bucket sort, the same set of numbers will be used to compare the two. For the implementation of parallel bucket sort, numbers were read in from a file and when it reached the point of k/n numbers, it was sent out to a processor. It is important to note that this action was not timed. The timer starts when each processor has its own big bucket and is ready to start putting their big bucket into small buckets. Once each processor has its small buckets, it sends them to the corresponding processor. Once each processor has received its small bucket, it sorts them using insertion sort. Insertion sort must be used in order to get an accurate comparison between sequential bucket sort and parallel bucket sort.



This surface shows the run time of different amounts of numbers, number of cores, and the time taken in seconds. As shown in the graph, when the amount of numbers are small, there is not much difference between the amount of cores. You start to see a difference between adding more cores at around 100000 numbers. After this point the amount of cores makes a huge difference in the run time.

Parallel vs. Sequential

The speedup of parallel against sequential was heavily dependent on the amount of numbers that had to be sorted. On the small side of the numbers tested, there was slowdown. This is because the parallel version is spending more time sending messages to and from different boxes. This is especially apparent when the number of cores goes up, as the number of cores goes up so does the run time. At around 50000 numbers there is speedup across the board, but not much for the higher number of cores. As the amount of numbers to be sorted goes up so does the speedup.

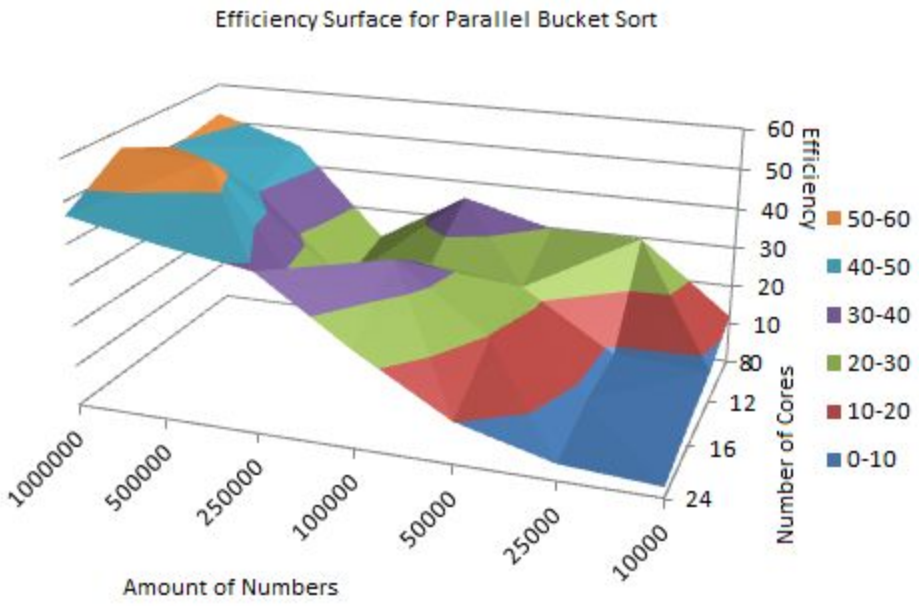


This surface shows the speedup of the parallel bucket sort against the sequential bucket sort. The small numbers produce slowdown, but as the numbers increase, so does the speedup. This chart below will provide the exact numbers.

Speed Up	8	12	16	24
10000	0.9586781474	0.8977750174	0.8616764164	0.6120478095
25000	2.370781985	1.044217771	0.9604503151	0.9608006229
50000	2.392510083	2.763256414	3.138716553	2.433374624
100000	2.853087938	3.169967543	5.738898577	5.644784538
250000	1.808185751	2.278959524	3.928183254	9.328877904
500000	3.624554366	4.821602682	8.793496243	10.16144559
1000000	4.166612734	5.947810457	8.93821459	11.19403816

Efficiency

The parallel bucket sort did not see great efficiency. The highest of which being 55.86%. This is due to the large amount of communication that must be done in order to pass all the small buckets around. Since there was a lot of initial communication, the time spent doing the actual work was reduced. However, if we used extremely large numbers, the efficiency would go up a lot since communication would only have to happen once.



This surface shows the Efficiency for the parallel bucket sort. 16 cores saw the highest efficiency with 1000000 numbers, but still only reached 55.86%. As expected, when the amount of numbers was small, efficiency was abysmal. This is because the processors were doing very little work compared to the communication they were doing.

Conclusion

Parallel bucket sort is much better than sequential bucket sort for a sufficient amount of numbers. I found that range to be about 100000 numbers for it to be worth it to parallelize. The communication is a big hindrance here, but with a machine that could use shared memory between a larger amount of cores would drastically improve the parallel bucket sort. Another way to improve this project would be to run larger data sets. Using insertion sort, it does not take long to reach the limit of five minutes of sequential running time. Using even larger numbers would show huge improvements in terms of speedup and efficiency especially using more cores. Overall, parallelizing bucket sort showed decent results. Bucket sort is already pretty efficient and so parallelizing does not show the greatest improvements. The efficiency

achieved was also not great. At a high of 55.86%, most processors were doing more communication rather than work. Therefore I have found that it is important to analyze things to make sure that parallelizing it would provide great benefits.