Adam Montano

CS 415 – Spring 2017

PA3 – Bucketsort: Final

Introduction

Bucketsort is a sorting algorithm that separates data into “buckets”, sorts each bucket individually and then merges the buckets back together. Bucketsort is best when the input data is uniformly distributed. The run time relies on the amount of input data and the number of buckets being used. In a parallel version, each processor can act as a bucket. Each bucket is sorted using a different sorting algorithm, usually insertion sort. In this implementation, each bucket was sorted with a bubble sort.

Sequential Implementation

The sequential implementation used a two-dimensional array with the second dimension acting as each bucket. The data was “scattered” to each bucket. Then, each bucket was sorted individually in place. Once each bucket was sorted, the data was “gathered” back into a single array. Finally, the data had been sorted. Below is a table of various sizes of inputs, buckets and the resulting run times. Also, vectors were used instead of arrays, which greatly increased the run time.

	4 Buckets	**8 Buckets**	**16 Buckets**	**24 Buckets**	**32 Buckets**
100 Numbers	4.6E-05	1.76E-05	1.53E-05	4.32E-05	1.45E-05
500 Numbers	0.000361	0.00022	0.000159	0.000141	0.000142
1000 Numbers	0.00135	0.0002	0.000403	0.000321	0.000158
5000 Numbers	0.015809	0.011249	0.006142	0.005543	0.002227
10000 Numbers	0.053268	0.028274	0.025163	0.010886	0.010156
50000 Numbers	1.06306	0.537672	0.276738	0.192771	0.139758
100000 Numbers	4.2303	2.11575	1.07002	0.71954	0.553945
800000 Numbers	267.996	120.256	52.6581	30.1493	18.8493

Here is this table graphed to a surface plot using Excel.

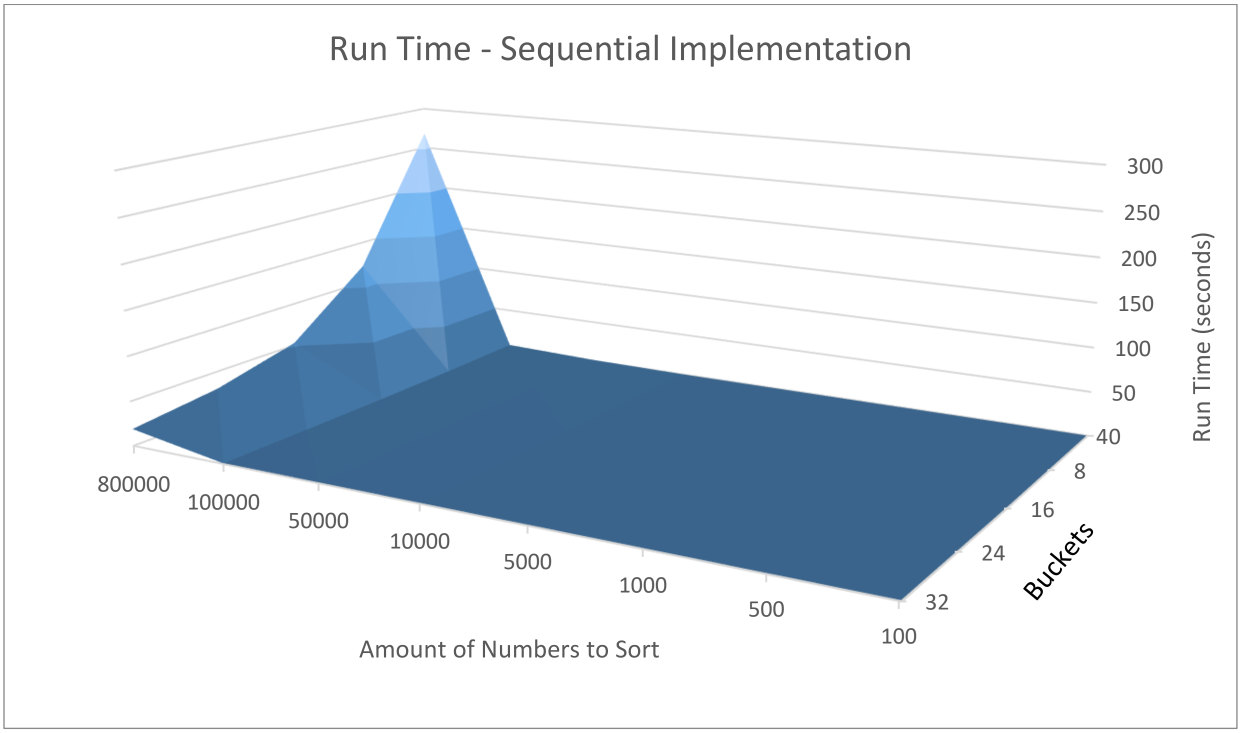


Figure 1 - Sequential Implementation

It is evident by the surface plot (Fig. 1) that an increased number of buckets allows for a general lower run time. The lower the number of buckets, the algorithm reverts closer to the sorting algorithm implemented for each bucket. A small amount of input gives near constant time regardless of the number of buckets. It is only when the input becomes fairly large is there a meaningful change in run time.

Parallel Implementation

The parallel implementation relies on a multicomputer system. Using MPI, the code forces each processor to receive a certain percentage of the input data that is to be sorted. Essentially, each processor acts as a bucket. Inside each processor, the numbers given are partitioned into "little buckets" and these little buckets are transferred to other processors. In effect, each processor now has some buckets that contain a certain interval of numbers. Each processor then runs a sorting algorithm (this implementation used bubble sort) on the received data. If the data is gathered back from each processor, all the numbers will be sorted.

Below is a table of data gathered from running this implementation with various sizes of processors, input data and the resulting run time.

	4 Buckets	**8 Buckets**	**16 Buckets**	**24 Buckets**	**32 Buckets**
100 Numbers	7.75E-05	0.00021	0.009464	0.020981	0.002174
500 Numbers	0.000127	0.000244	0.107984	0.001953	0.00246
1000 Numbers	0.000288	0.000145	0.001736	0.024088	0.001758
5000 Numbers	0.008902	0.003596	0.002023	0.005555	0.00151
10000 Numbers	0.01841	0.004132	0.084693	0.024526	0.002321
50000 Numbers	0.279388	0.067782	0.01965	0.021709	0.026769
100000 Numbers	1.07471	0.282469	0.069544	0.0501	0.043935
800000 Numbers	67.561	16.6846	4.15893	1.83836	1.05061

Below is a surface plot of the data from this table.

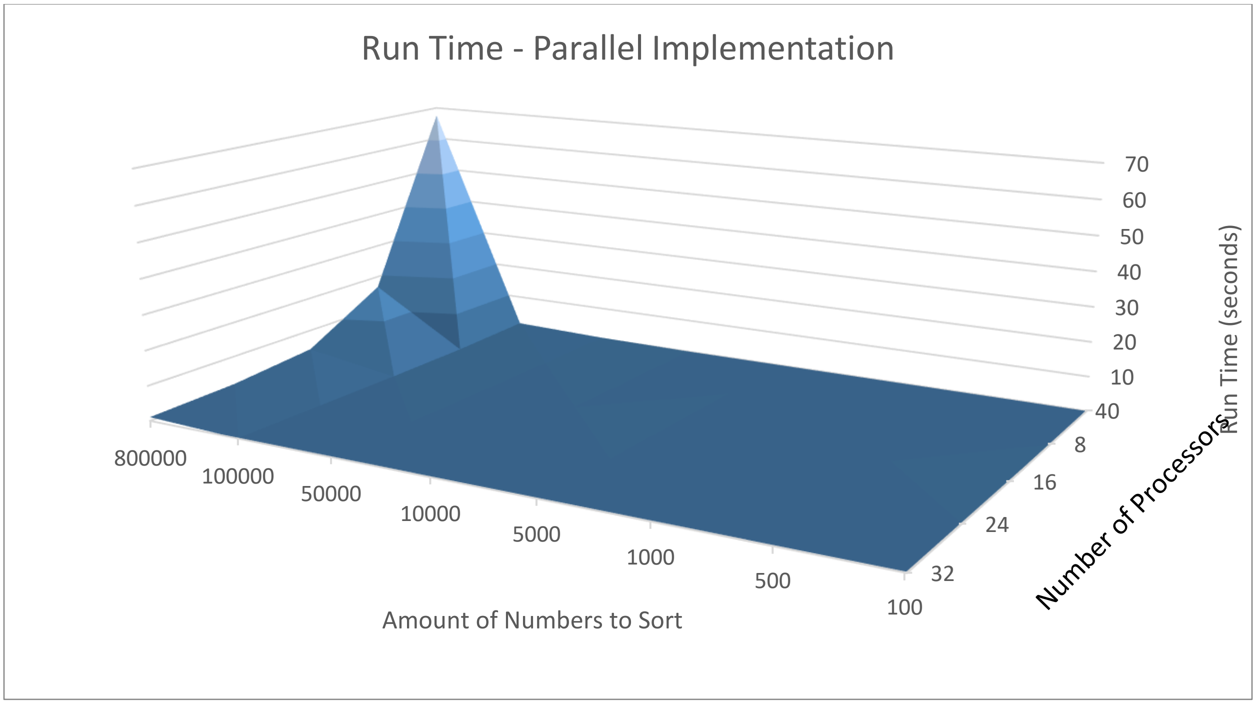
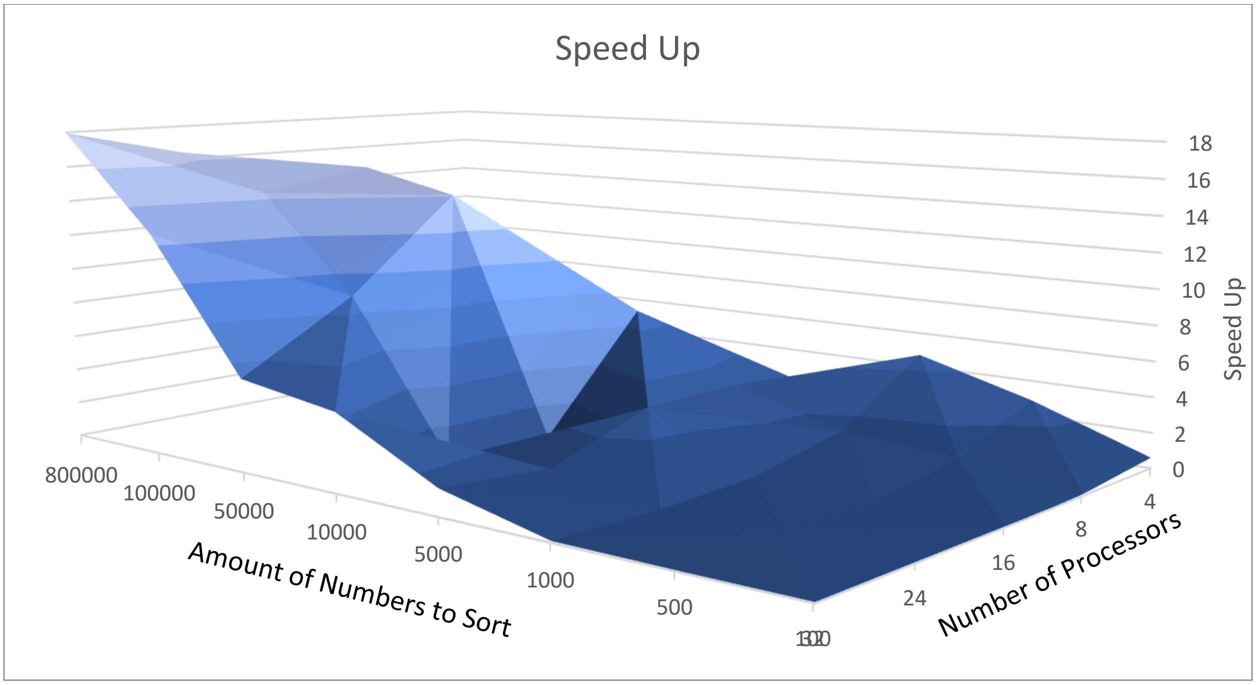


Figure 2 - Parallel Implementation

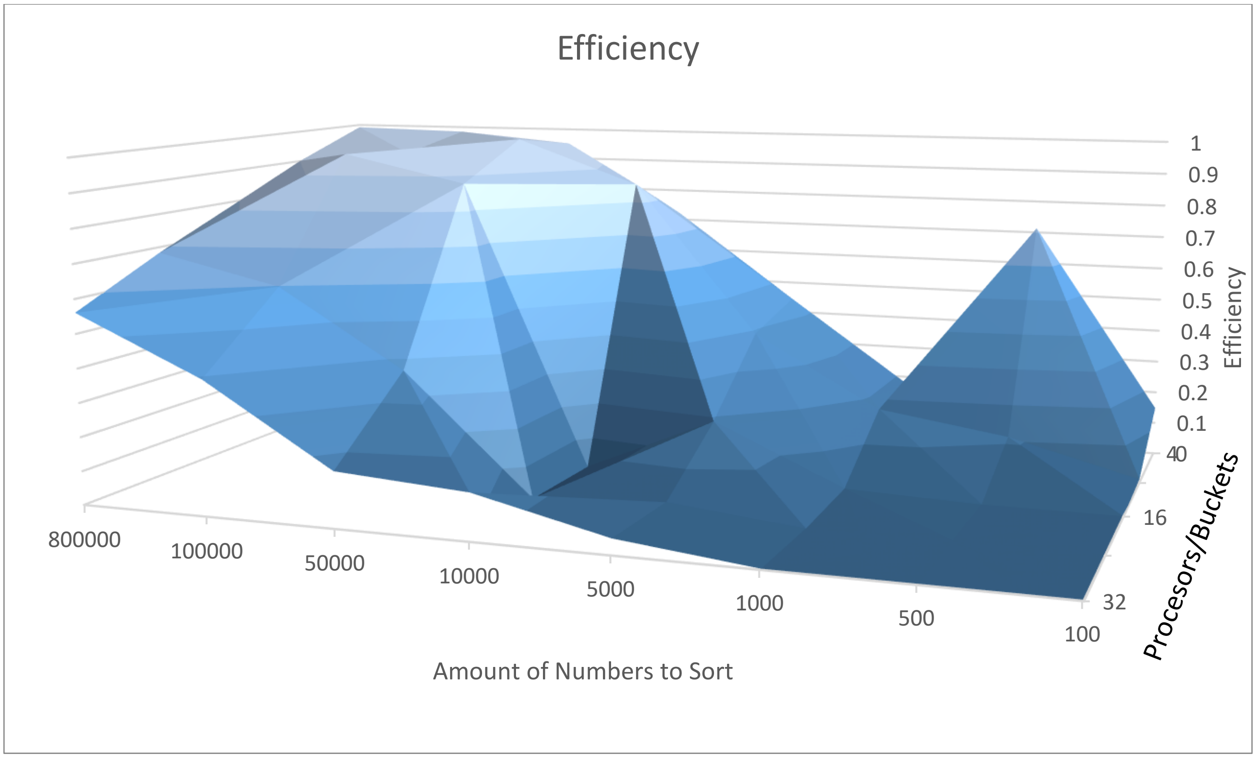
The parallel implementation has an overall similar shape to the sequential implementation. There is very little meaningful change until a large amount of data is processed. This limit must be where the thrashing begins. However, the parallel time has a much smaller overall run time for each size of data and number of buckets used. It's clear that a greater number of buckets has a dramatic decrease in the run time. While testing, the implementation was very sensitive to the increased size of input data when the size was nearing one million numbers.

Analysis and Comparisons

Overall, the speed up was large. As expected, the speed up is stronger with a greater number of processors and a large amount of data. When the data was small, there was less significant speed up. The maximum speed up was nearly 18 with 32 processors. Below is a surface plot that represents the speed up calculated from the earlier tables.



The efficiency was also calculated from the table and graphed below. The max efficiency was very high, almost reaching 100% but there are many dips and spikes on the graph. The valley experienced where there are 4 buckets and 1000 numbers to sort is most likely due to the network congestion. As there is more data to process, the efficiency rises again. When there is a low amount of data but a large number of buckets/processors, the system is not used very efficiently at all. Most of the computation is completed very quickly but the communication would dominate the total run time.



Issues

The implementation of this program could have been much easier had it been developed using vectors instead of arrays due to the nature of the method of scattering. So, the sequential was updated to use vectors instead of arrays. However, vectors are much slower than arrays as they guarantee contiguous memory and have more maintenance than normal C style arrays. As such, the run time was greatly increased. I would have liked to show more data but sequential time capped at 5 minutes when processing more than eight hundred thousand numbers. Also, the parallel implementation was difficult to complete. There were many memory errors that took a large amount of time to debug.

Conclusion

The parallel implementation greatly dominated in decreasing the run time even though each bucket was being sorted with bubble sort, a notoriously slow sorting algorithm. I would have liked to show more data but the sequential implementation capped at 5 minutes. Also, there were lots of issues when trying to use vectors with MPI but it was resolved when converting the vectors to arrays.