Kurt Andersen

CS 615

PA3 Bucket Sort

April 6, 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. The code was then parallelized and timings were done for the beginning and each of each box. I decided to use these numbers of processors because it shows the jump between each box.

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 labeled with how many data entries were used.

For the parallel portion I used essentially the same algorithm. But to attempt to avoid blowing up the message buffer, as I read in data, I sent it out to each processor as a chunk was read in. Each processor then waited at a barrier before starting the actual algorithm. Once all processes hit the barrier, they began distributing their data into buckets. Those buckets were then distributed to the appropriate processor. Once all the processors received their new data from the previously organized buckets, they sorted all the data they have. After the sorting, the processes waited at a barrier again for accurate timing. Once all processes hit the barrier the data was all sorted and the timer would stop.

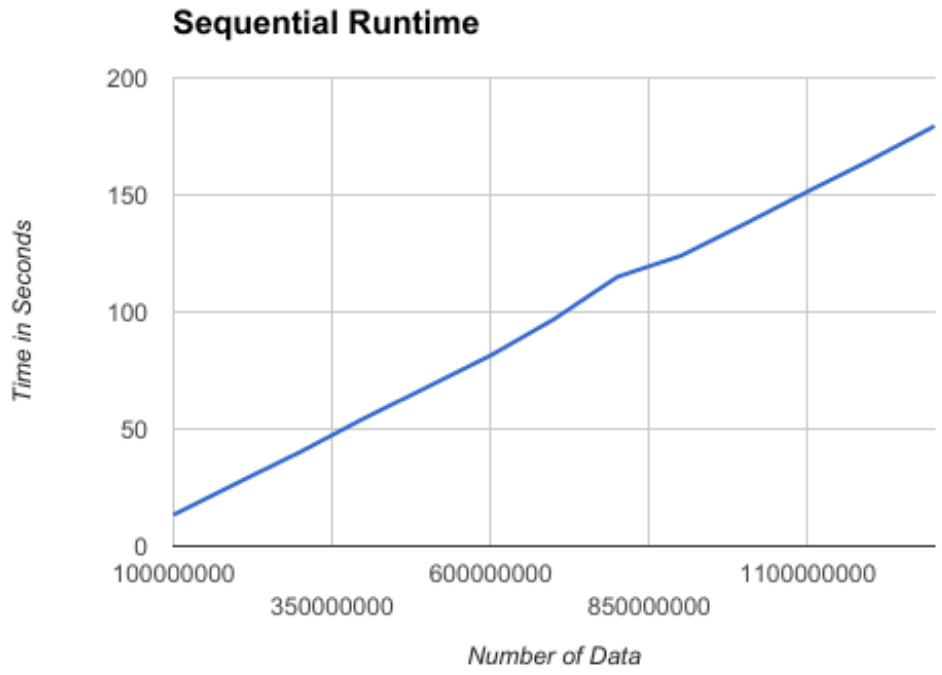
The only issue I ran into for the sequential portion was when I was calculating the min and max of each bucket. It was not even a programming problem, but just more of an issue on my math logic. I knew what I wanted in my head, i.e. the first bucket would be from 0 to 1/10 of the max value, but when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

For issues in the parallel portion, the biggest one I had was devising a message passing scheme. What I had originally tried was having each processor just cram all of its data into the buffer, then all the processors would try and gather it all. This ended up blowing the entire buffer up and the program would hang. I used a similar idea, but in the loops, if I was the current processor for the loop iteration, then I would be receiving all the message. If I wasn't the current processor then I would be sending my data to the iterator process. This massively reduced the amount of data being pushed into the buffer.

Results:

I would say my overall results for the bucket sort project was successful. All the following data supports and proves that bucket sort is not meant to be parallelized. This is due to the fact that bucket sort is an nlogn time complexity. It is extremely difficult to go beyond the nlogn complexity when parallelized. Parallelization is best used for algorithms that have an n^2 complexity or greater. The individual data results can all be found within the bin folder of the project directory. For the next project I need to learn how to get all of my data printed to an xml file instead of copy pasting from 100+ files into an xml worksheet.

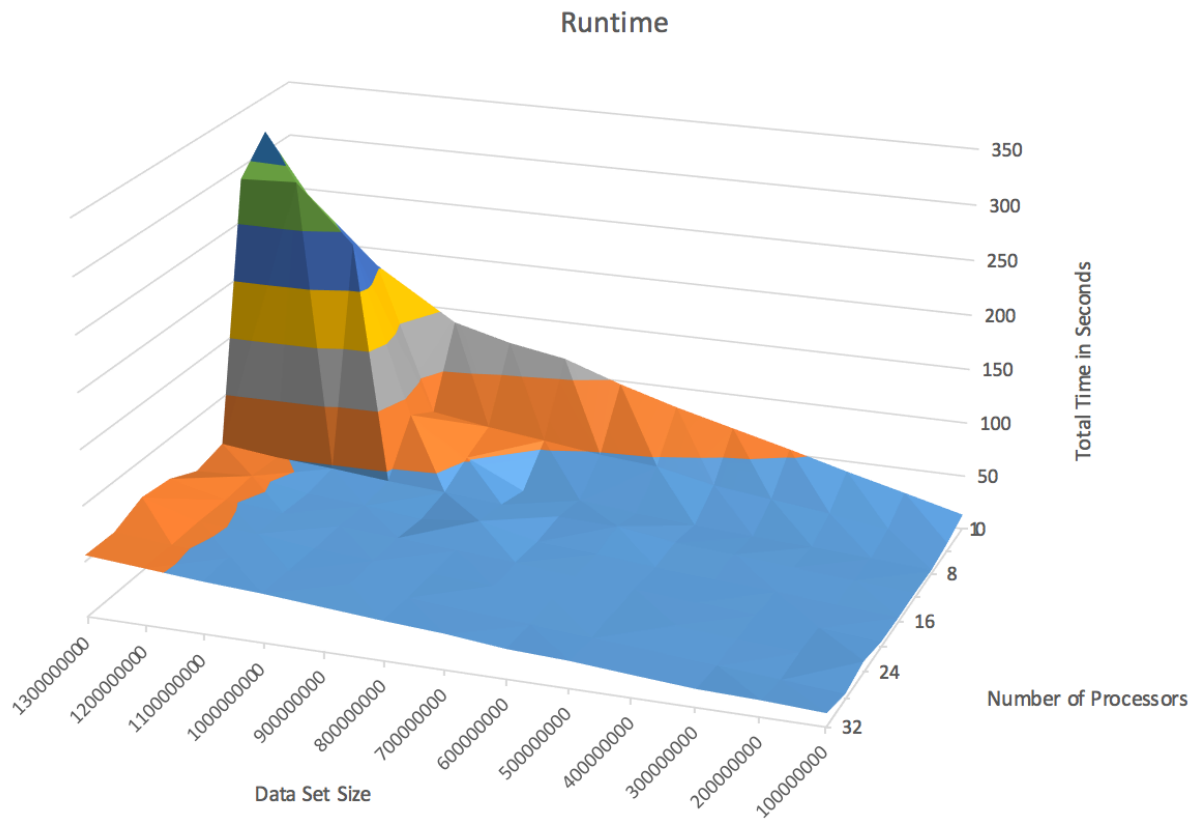
Sequential Run Time			
Data In Set	Time	Data In Set	Time
100000000	13.344	700000000	96.92
200000000	26.947	800000000	114.993
300000000	40.231	900000000	123.99
400000000	54.5245	1000000000	137.5
500000000	67.895	1100000000	151.413
600000000	81.392	1200000000	165.024
		1300000000	179.372



In the above graph, this shows the sequential runtime of bucket sort. The graph shows to be linear since I was using the std library sort. As we linearly increased the values in the graph, we linearly increased the time in which it took to execute the overall sort. Bucket sort is an algorithm that is almost designed to be run sequentially. The amount of speedup and the efficiency of running on multiple processors isn't very good at all. This is an algorithm that is more designed for sequential running. If this algorithm was more focused on a divide and conquer method, it could possibly be done much faster in parallel.

Overall Parallel Runtime									
Data In Set	Number Of Processors								
	1	2	8	9	16	17	24	25	32
100000000	13.344	7.078	3.3301	4.271	3.817	4.586	10.215	4.762	11.923
200000000	26.947	13.3407	6.825	8.578	7.717	9.3322	10.6331	8.936	14.6049
300000000	40.231	20.83976	10.116	12.711	11.7874	12.856	14.192	13.322	16.074
400000000	54.5245	27.1444	12.376	17.298	15.267	17.068	18.545	17.575	20.031
500000000	67.895	33.339	15.828	21.006916	18.841	20.7759	21.1758	21.678	24.26
600000000	81.392	44.2111	19.783	28.596	23.984	25.577	26.206	25.5966	26.5794
700000000	96.92	48.4473	24.924	30.4394	27.614	29.858	30.489	29.614	32.052
800000000	114.993	55.178	26.6888	34.37	39.607	33.784	33.4879	33.979	35.535
900000000	123.99	61.4271	51.4838	38.836	34.176	37.454	38.0007	39.0229	39.818
1000000000	137.5	69.4752	85.416	42.28	39.476	41.519	41.671	43.053	44.052
1100000000	151.413	203.4988	239.363	47.189	43.655	45.945	47.868	47.469	47.178
1200000000	165.024	245.5754	291.505	50.77	53.109	51.068	52.005	51.1196	51.4906
1300000000	179.372	318.1347	289.1149	56.8599	50.58	63.358	66.702	55.225	55.656

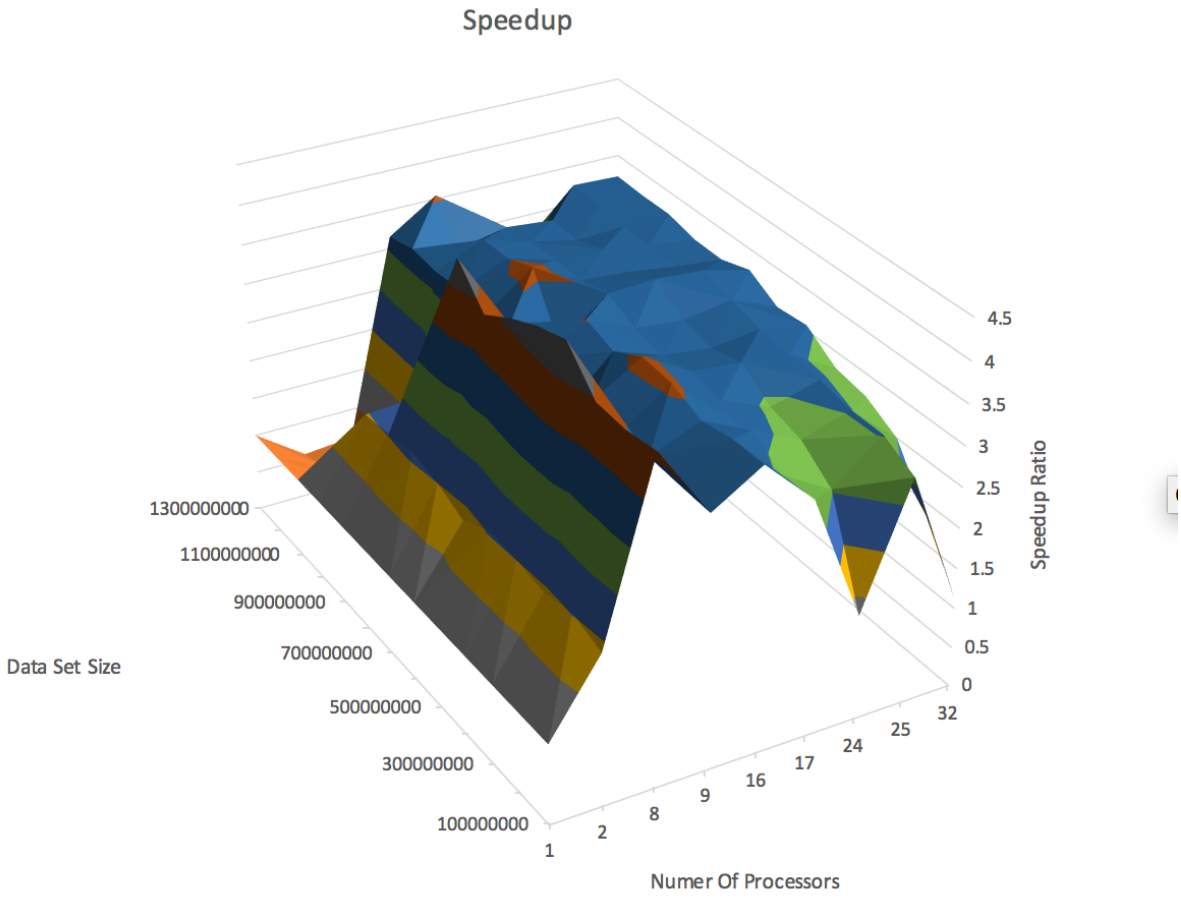
Runtime graph shown on next page



This graph shows the overall runtime comparing the size of the data set and the number of processors used. This graph shows the an almost linear growth for each of the number of processors. There was a spike in the timing around 1 billion data points in the smaller number of processors due to thrashing occurring when messages would be passed. Aside from the thrashing, the graph holds strong to what was expected prior to running the code. The graph shows that obviously when run in parallel, it takes less and less time, but how does this look with the efficiency on the processors?

Speedup Values									
Data Set Size	Number Of Processors								
	1	2	8	9	16	17	24	25	32
100000000	1	1.885278327	4.007086874	3.124326856	3.495939219	2.909725251	1.306314244	2.802183956	1.119181414
200000000	1	2.019909	3.948278388	3.141408254	3.491900998	2.8875292	2.534256238	3.015555058	1.845065697
300000000	1	1.930492482	3.976967181	3.16505389	3.413051224	3.129355943	2.834766065	3.019891908	2.502861764
400000000	1	2.008683191	4.405664189	3.152069603	3.571395821	3.194545348	2.94011863	3.102389758	2.722005891
500000000	1	2.036503794	4.289550164	3.232030823	3.603577305	3.267969137	3.206254309	3.13197712	2.798639736
600000000	1	1.840985635	4.114239499	2.846272206	3.39359573	3.182234038	3.105853621	3.179797317	3.062221119
700000000	1	2.000524281	3.888621409	3.184031223	3.509813863	3.246031214	3.178851389	3.27277639	3.023836266
800000000	1	2.084037116	4.308661311	3.345737562	2.903350418	3.403771016	3.433867158	3.384237323	3.236048966
900000000	1	2.01849021	2.408330387	3.192656298	3.627984551	3.310460832	3.262834632	3.177365086	3.113918328
1000000000	1	1.979123486	1.609768662	3.252128666	3.48312899	3.311736795	3.299656836	3.193737951	3.121311178
1100000000	1	0.7440486136	0.6325664368	3.208650321	3.468399954	3.295527261	3.163136124	3.18972382	3.209398448
1200000000	1	0.6719891325	0.5661103583	3.250423478	3.107269954	3.231456098	3.173233343	3.228194274	3.204934493
1300000000	1	0.5638240657	0.6204176955	3.154630944	3.546302887	2.83108684	2.689154748	3.248021729	3.222869053

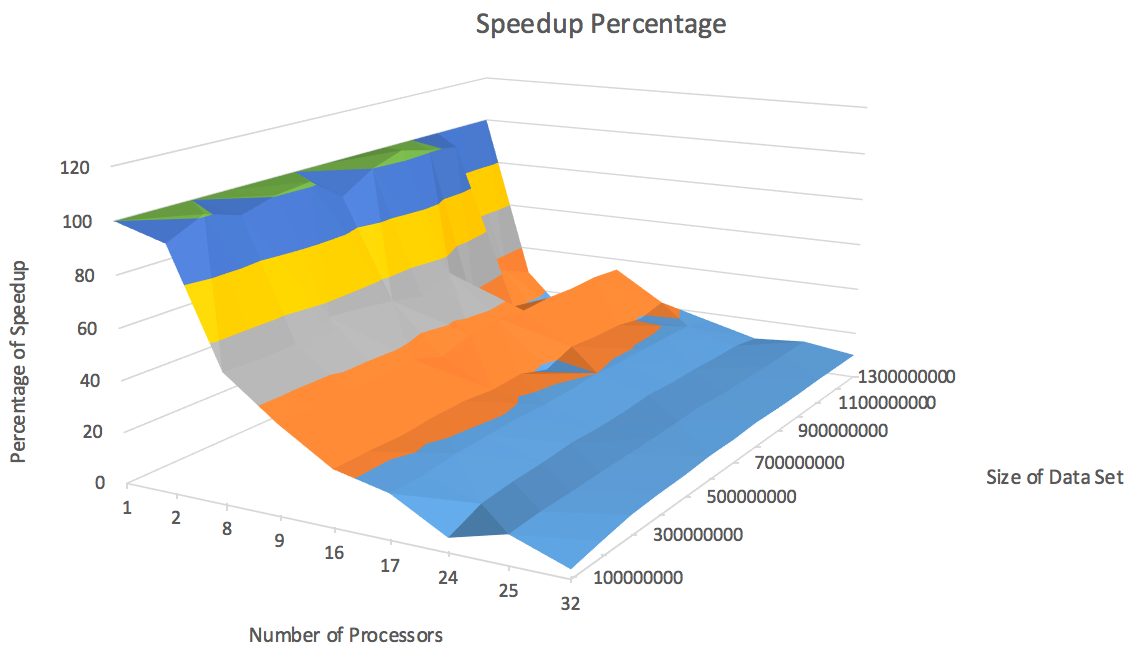
This data table refers to the calculated speedup for each overtime run in parallel. The column with 1 processor all reads 1, because there is no speed up. The speedup is calculated as the sequential time divided by the parallel time.



This graph represents the comparison of the data set and the number of processors being used to determine the overall speedup. The greatest amount of speedup is when we used a minimal amount of processors. The highest actual speedup occurs right before the calculations are moved to an additional box. This is due to the increased communication times between the boxes. At approximately 8 processors, we experience the best speedup. This is due to a well balanced ratio of communication and message passing time. We can see that as there is an increased amount of processors, the speedup factor drops because we are spending more and more time passing messages than actually doing calculations.

Efficiency Percentage									
Data Set Size	Number of Processors								
	1	2	8	9	16	17	24	25	32
100000000	100	94.26391636	50.08858593	34.71474284	21.84962012	17.11603089	5.442976016	11.20873583	3.497441919
200000000	100	100.99545	49.35347985	34.90453615	21.82438124	16.98546588	10.55940099	12.06222023	5.765830304
300000000	100	96.52462408	49.71208976	35.16726545	21.33157015	18.40797613	11.81152527	12.07956763	7.821443014
400000000	100	100.4341595	55.07080236	35.02299559	22.32122388	18.79144322	12.25049429	12.40955903	8.506268409
500000000	100	101.8251897	53.61937705	35.91145358	22.52235816	19.22334787	13.35939295	12.52790848	8.745749176
600000000	100	92.04928174	51.42799373	31.62524673	21.20997332	18.71902376	12.94105676	12.71918927	9.569440996
700000000	100	100.0262141	48.60776761	35.3781247	21.93633664	19.09430126	13.24521412	13.09110556	9.449488331
800000000	100	104.2018558	53.85826639	37.1748618	18.14594011	20.02218245	14.30777982	13.53694929	10.11265302
900000000	100	100.9245105	30.10412984	35.47395887	22.67490344	19.47329901	13.5951443	12.70946034	9.730994776
1000000000	100	98.95617429	20.12210827	36.13476296	21.76955619	19.48080468	13.74857015	12.7749518	9.75409743
1100000000	100	37.20243068	7.907080459	35.65167023	21.67749971	19.38545448	13.17973385	12.75889528	10.02937015
1200000000	100	33.59945662	7.076379479	36.11581643	19.42043721	19.00856528	13.2218056	12.9127771	10.01542029
1300000000	100	28.19120329	7.755221194	35.05145493	22.16439304	16.653452	11.20481145	12.99208692	10.07146579

This chart shows the overall efficiency from our previous calculations. Efficiency is achieved by taking the speedup factor obtained before, and dividing it by the number of processors used. Any efficiency over 100 is good, because that means we are speeding up enough to have a cost effective algorithm to keep adding processors. The lower the number, the more costly your algorithm is.



*Typo on graph name. Should be “Efficiency Percentage”

This graph shows the efficiency percentage of the previously calculated data. As we can see, even though we had a decent speedup factor around 8 cores, the overall efficiency is not worth the time. This graph is showing how inefficient bucket sort is when run in parallel We are actually wasting more time using more cores for bucket sort because bucket sorts normal run time is nlogn. It is extremely difficult to go beyond nlogn for a sorting method.

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. The knowledge gained from this project can be applied to knowing what kind of algorithms are best designed to be parallelized. The best example will be our future project of doing matrix multiplication which has an n^3 time complexity.