PA3: Bucket Sort

Evan Grill

CS 415

April 6, 2017

Problem

This project asked us to implement a bucket sort both sequentially and in parallel.

Procedure

For this version of the project, rather than generate test sets of data beforehand, we instead had our programs generate data at runtime. In the sequential version, the full list is generated and then bucket sort performed. The timer timed only the bucket sort. In the parallel version, the data for each node is generated and passed to each of the nodes, including the master node. The parallel nodes would distribute their data set into buckets and then the buckets would be passed to the node that would handle that range of data. In both versions, the final buckets were sorted using std::sort as it is a reliably fast sorting implementation.

Data

	Data Size										
# Tasks	10	100	1000	10000	100000	1M	10M	100M	1B	1.5B	2B
1	0.00001	0.00003	0.00006	0.00165	0.01313	0.05035	0.53230	6.07698	64.87220	107.50000	140.10000
2	0.00008	0.00018	0.00032	0.00128	0.00401	0.04054	0.44876	4.68007	48.37470	77.00550	100.90000
4	0.00010	0.00019	0.00034	0.00067	0.00550	0.02140	0.23983	2.54412	25.75640	40.36380	51.63290
8	0.00104	0.00080	0.00131	0.00158	0.01010	0.00986	0.12743	1.31520	14.64010	91.43750	227.71600
16	0.00299	0.00068	0.00644	0.00064	0.00198	0.00648	0.14697	1.52228	17.05990	44.13690	31.62740
24	0.03821	0.00158	0.00217	0.00145	0.00172	0.00800	0.23217	1.77121	18.40720	27.69410	36.49920
32	0.01253	0.01142	0.00395	0.00477	0.00435	0.00552	0.23477	1.72943	17.39050	25.47470	33.33410

Table 1: Execution time (in seconds) of Sequential and Parallel Bucket Sort.

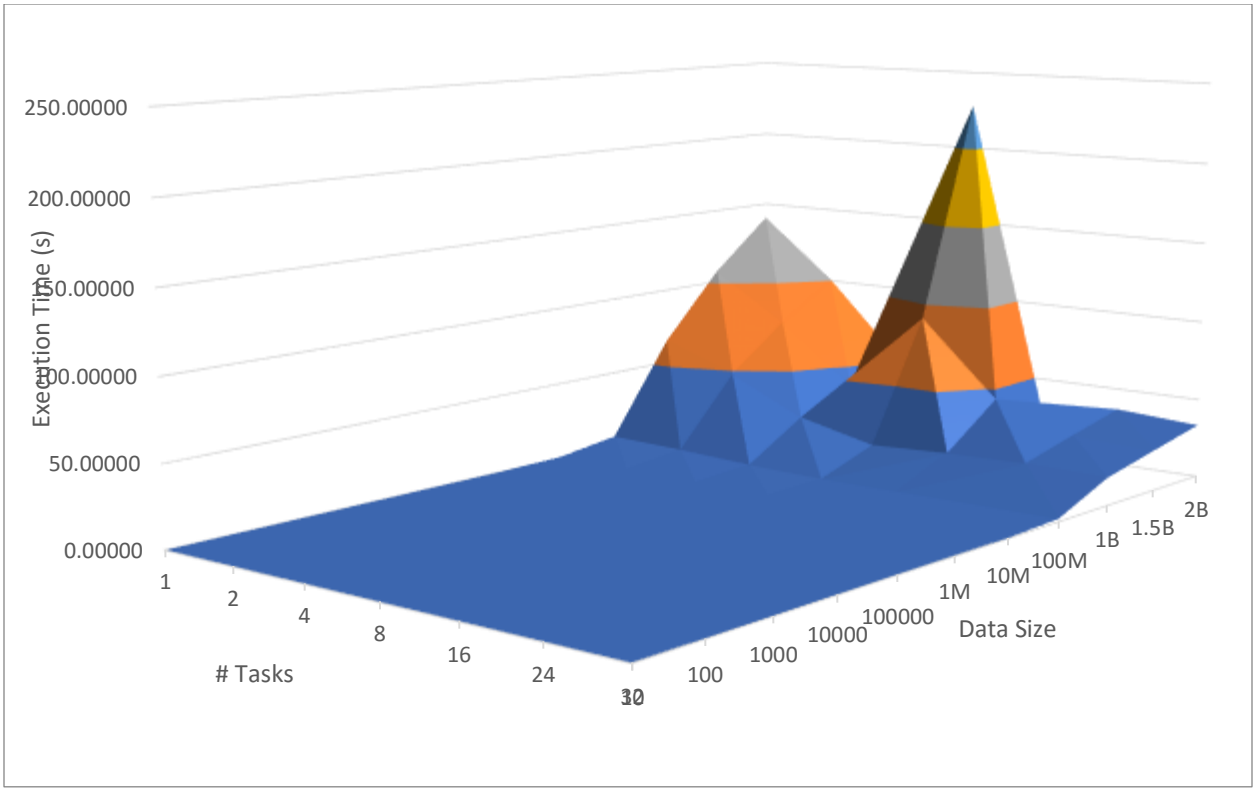

Figure 1: Chart showing execution times from Table 1.

	Data Size										
# Tasks	10	100	1000	10000	100000	1M	10M	100M	1B	1.5B	2B
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	0.15430	0.13583	0.17682	1.28728	3.27281	1.24182	1.18614	1.29848	1.34104	1.39600	1.38850
4	0.11927	0.12947	0.16491	2.47726	2.38577	2.35206	2.21946	2.38864	2.51868	2.66328	2.71339
8	0.01193	0.03119	0.04264	1.04017	1.29983	5.10573	4.17706	4.62057	4.43113	1.17567	0.61524
16	0.00414	0.03697	0.00871	2.57986	6.62915	7.77050	3.62174	3.99203	3.80261	2.43560	4.42970
24	0.00032	0.01589	0.02586	1.13400	7.61348	6.29192	2.29267	3.43098	3.52428	3.88169	3.83844
32	0.00099	0.00219	0.01420	0.34549	3.01612	9.12624	2.26727	3.51386	3.73032	4.21987	4.20290

Table 2: Speedup factor for parallel bucket sort vs. sequential bucket sort.

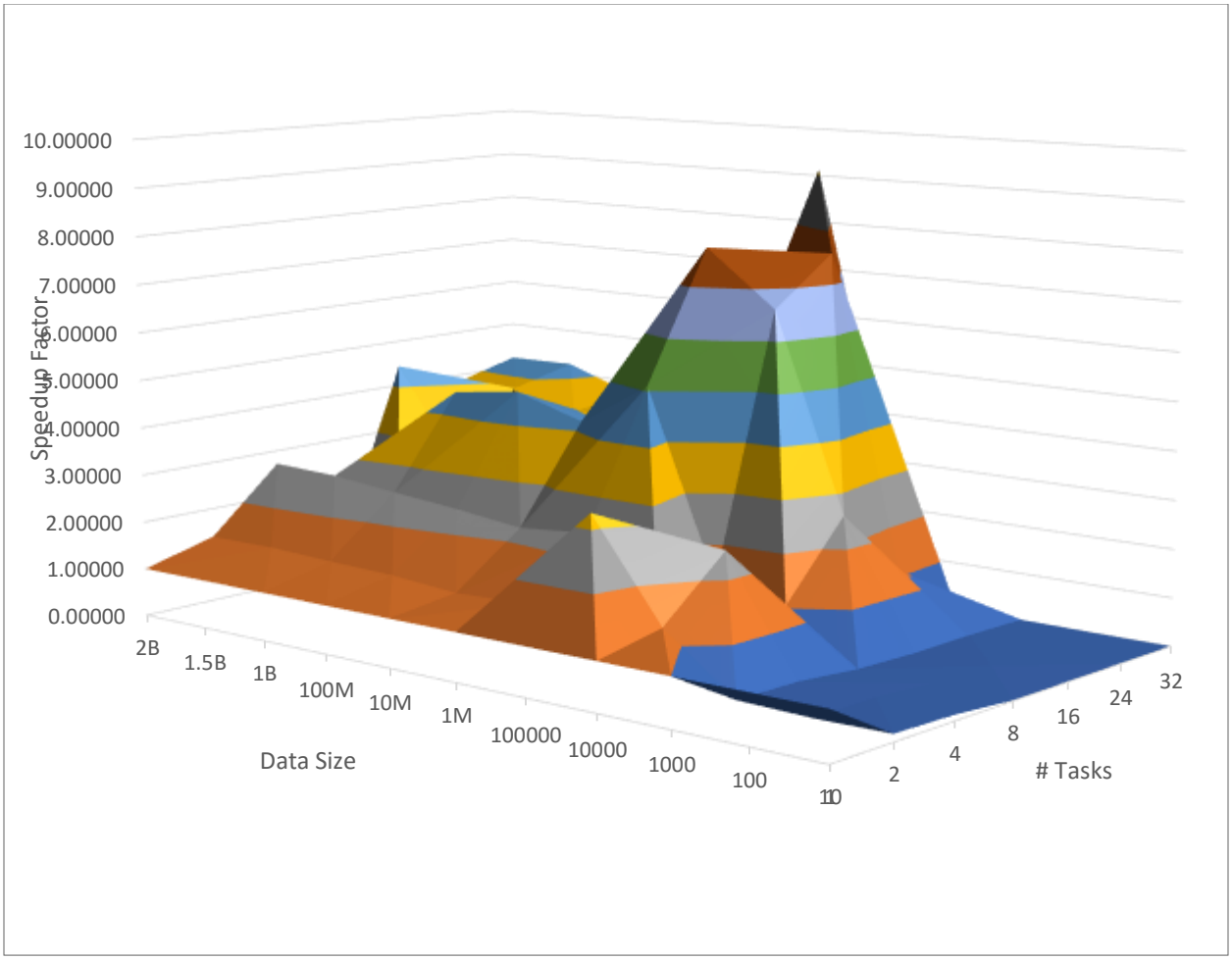

Figure 2: Chart showing speedup data from Table 2.

	Data Size										
# Tasks	10	100	1000	10000	100000	1M	10M	100M	1B	1.5B	2B
1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
2	7.72%	6.79%	8.84%	64.36%	163.64%	62.09%	59.31%	64.92%	67.05%	69.80%	69.43%
4	2.98%	3.24%	4.12%	61.93%	59.64%	58.80%	55.49%	59.72%	62.97%	66.58%	67.83%
8	0.15%	0.39%	0.53%	13.00%	16.25%	63.82%	52.21%	57.76%	55.39%	14.70%	7.69%
16	0.03%	0.23%	0.05%	16.12%	41.43%	48.57%	22.64%	24.95%	23.77%	15.22%	27.69%
24	0.00%	0.07%	0.11%	4.73%	31.72%	26.22%	9.55%	14.30%	14.68%	16.17%	15.99%
32	0.00%	0.01%	0.04%	1.08%	9.43%	28.52%	7.09%	10.98%	11.66%	13.19%	13.13%

Table 3: Efficiency of parallel bucket sort vs. sequential bucket sort.

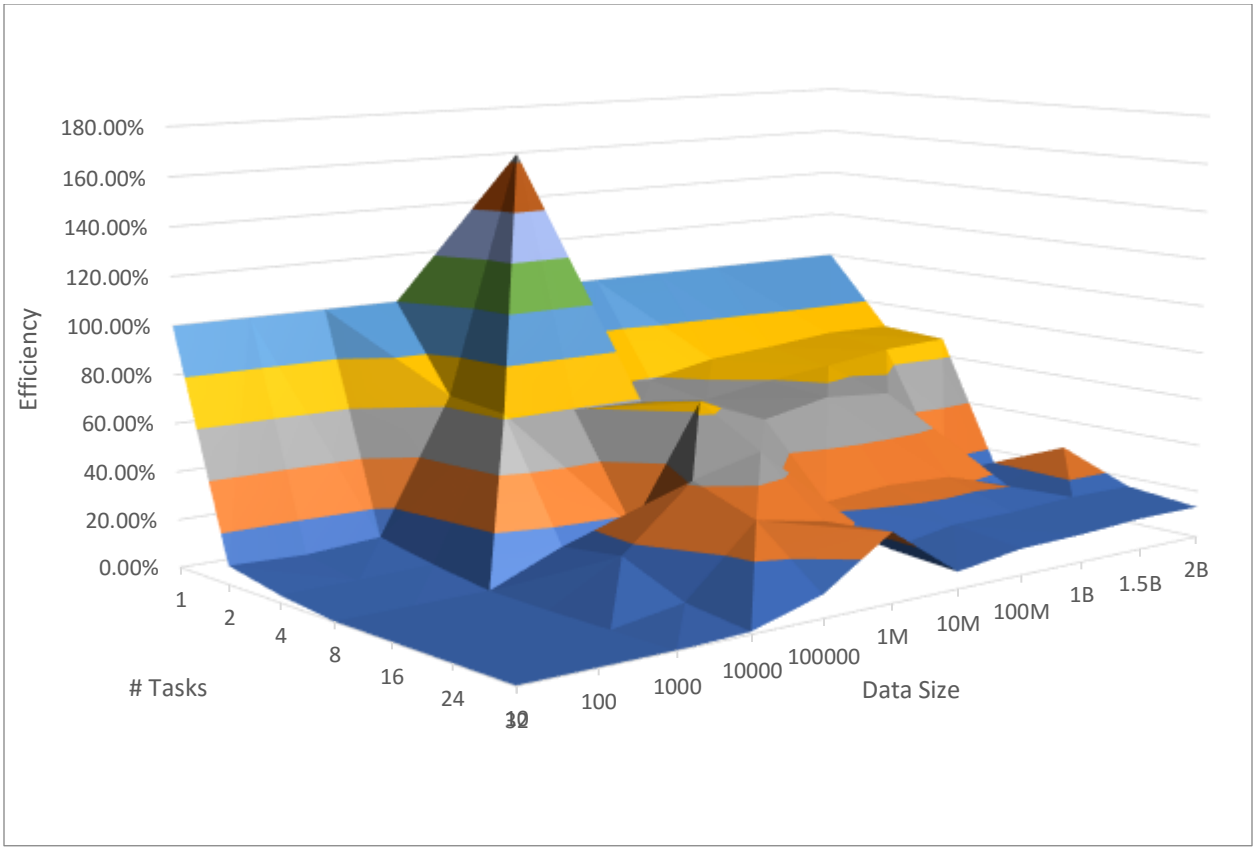

Figure 3: Chart representing efficiency data from Table 3.

Results

As shown in the Data section, execution time decreased with the addition of cores and speedup increased, although not in a wholly linear fashion. Rather than verbally synthesize the above results, this section will discuss the interesting data points.

In Figure 1, you can see a large spike in execution time with 8 cores operating on a data set of 2 billion. At first I thought this was a strange anomaly, but it occurred even after running the same parameters multiple times. My only guess to the sudden increase in execution time is that the program as implemented is causing thrashing with these specific parameters. Given more examination and debugging, this could likely be fixed.

Next, in Figure 2, you can see the maximum speedup was with 32 cores operating on a data set of 1 million. More interestingly, you can see using 2 cores with a data set 100 thousand gave us superlinear speedup (greater than 2). These results were again shown even with multiple test runs with these parameters. My intuition is that while the parallel algorithm differs only slightly from the sequential one, in this instance it is much more efficient than one would expect.

Continuing, in Figure 3, you can see the corresponding spike in efficiency that comes with the superlinear speedup we saw in Figure 2. Along with that we can see that in general, the

parallel algorithm is most efficient with data sets in the 100 thousand to 10 million range. It is also most efficient with 2 to 8 cores, as increasing the number of cores beyond that range requires transferring large amounts of data (gigabytes) across network connection rather than just in internal memory.

Lastly, I'd like to talk about the differences in execution times in this report versus the report for part 1 (sequential only). In the intervening time, I improved the memory handling of the sequential algorithm by reducing the amount of copies of data and that improved performance greatly.