Andrew Frost

CS415

PA3 Report

04/05/2017

Overview:
In Programming Assignment 3, a sequential bucket sort and a parallel bucket sort was implemented and tested on a large number of integers. The underlying sort used to sort each bucket was heap sort with a time complexity of O(n logn). In this case sets of 10 to 1,000,000,000 integers were randomly generated and sorted by the algorithm. All of the sorted integers fall between 0 and 100,000. The times were recorded only for the sorting portion of the program. File I/O and number generation is not accounted in the timing results. Multiple trials were conducted for each number of integers sorted. The results of these trials were averaged to create the timing data discussed in this report. Not every number of integers between 10 and 1,000,000,000 integers were tested; only select benchmark numbers of integers were sorted for the purpose of ensuring the efficient use of cluster resources. The complete data for this report can be found in data.xlsx. Table 1 displays a sample of 25 randomly generated numbers in an unsorted and sorted state.

25 Random Integers Unsorted and Sorted

Unsorted	Sorted
383	27
886	59
777	172
915	211
793	335
335	362
386	368
492	383
649	386
421	421
362	426
27	429
690	492
59	540
763	567
926	649
540	690
426	736
172	763
736	777
211	782
368	793
567	886
429	915
782	926

Table 1: a 25 sample set of randomly generated integers between 0 and 999 in its unsorted and sorted forms.

Table 1 displays an example set of data that was sorted by the sequential bucket sort algorithm. The sorted data output by the program is organized in the same way as the input data where the number of pieces of data to sort is followed by the data. The sequential bucket sort algorithm’s run time will now be examined.

Sequential Bucket Sort:
The sequential bucket sort algorithm made use of 100 buckets. As the data was processed minimum and maximum values were recorded to determine the interval the data was distributed over. To sort the data, each integer was divided by a partitioning value that would yield the integer's target bucket. If any bucket contained more than 1 integer, then the bucket was sorted using a heap sort method which runs in O(n log(n)). Timing results were taken at intervals for efficiency purposes. A graph of the timing results of the sequential bucket sort algorithm is displayed in Fig. 1.

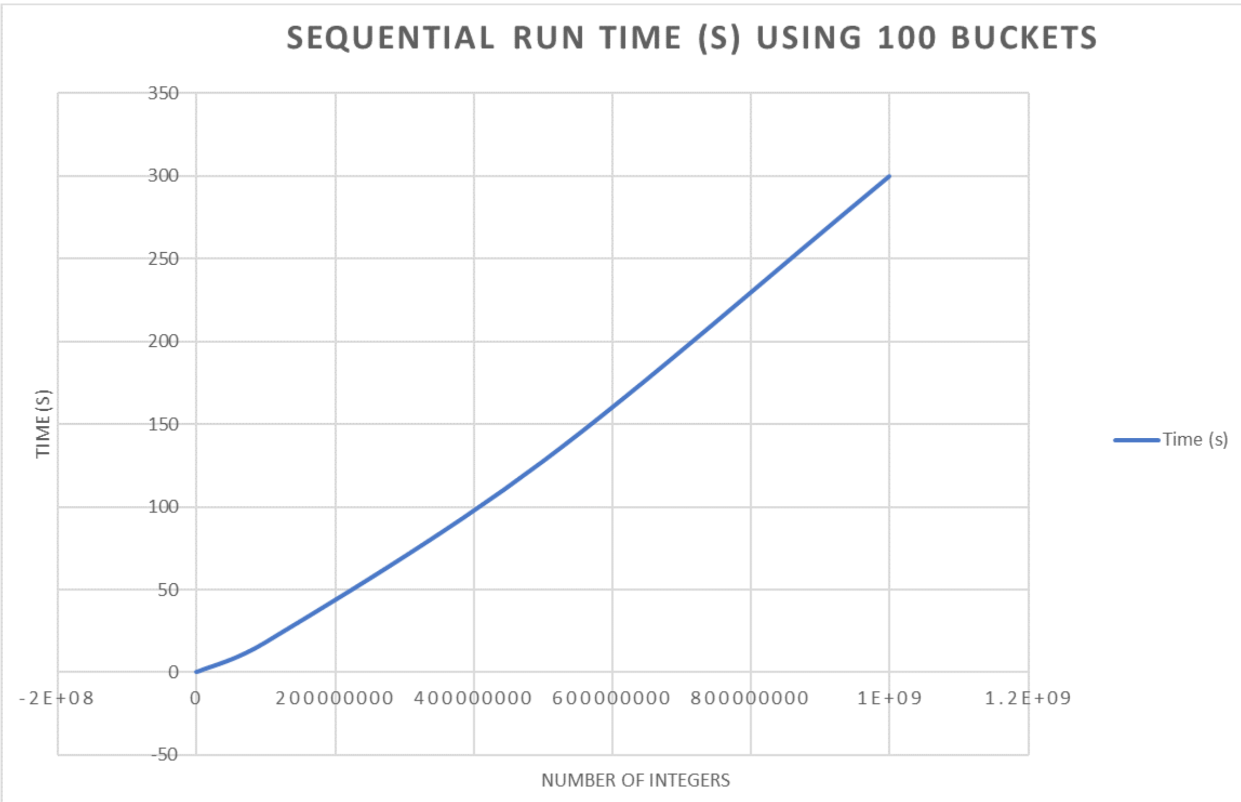

Figure 1: A graph of the averaged timing results of sorting 10 to 1,000,000,000 integers. The times displayed are in seconds

As the graph in Figure 1 shows, the time to sort an increasing number of integers increased slightly greater than linearly. The range between 100,000,000 and 1,000,000,000 saw a slightly sharper increase in comparison to the rest of the sample data. In the event of a poor partitioning into buckets, the sort's efficiency would begin to approach the efficiency of the sort used to sort individual buckets. The averaged timing results are displayed in Table 2.

The Averaged Timing Results of Sorting Integers Sequentially

Number of Integers	Time (s)
10	1.4E-05
100	4.96E-05
1000	0.0002308
10000	0.001913
100000	0.0124338
1000000	0.126415
10000000	1.51225
100000000	17.9251
500000000	127.31
1000000000	299.778

Table 2: The average time to required to sort different numbers of randomly generated integers between 0 and 100,000. The number of integers to be sorted is over a range from 10 to 10,00,000,000 integers. Note the massive jump in sorting times between 100,000,000 elements and 1,000,000,000 elements.

As the number of integers to be sorted increased so did the amount of time required to sort them. The parallel algorithm will now be compared to the sequential algorithm and discussed.

Parallel Bucket Sort:
The parallel bucket sort algorithm had a bucket for each core used. Each bucket was sorted using the same heap sort method as used in the sequential. This bucket sort algorithm required a lot of synchronization time, which increased the communication time and latency. Despite its limitations, the parallel algorithm performed surprisingly well. Figure 2 shows the run time of the parallel algorithm using 2, 3, 4, 5, 8, 9, 16, 17, 24, 25, 32, and 33 cores and buckets.

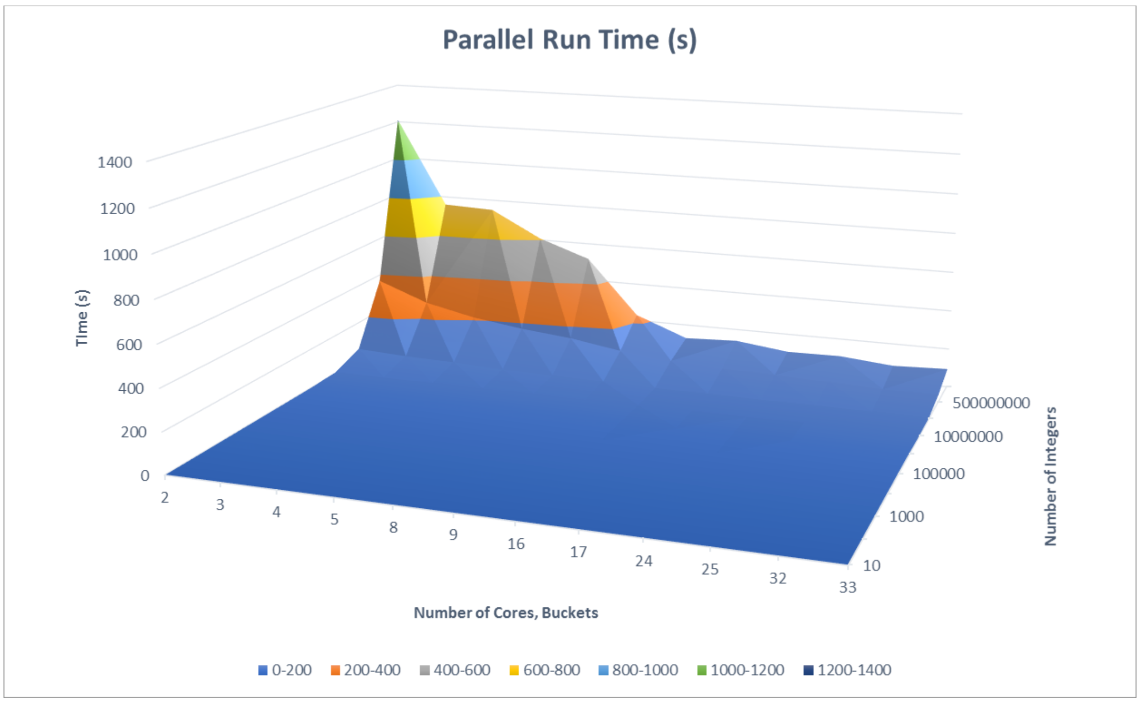

Figure 2: The run time in seconds of the parallel algorithm across 2, 3, 4, 5, 8, 9, 16, 17, 24, 25, 32, and 33 cores and buckets over a range of 10 to 1,000,000,000 integers.

Consistently, as the number of cores increased the run time of the algorithm decreased as more work was delegated between more processors. In the case of using less than 24 cores when sorting 500,000,000 to 1,000,000,000 integers, the amount of run time required to sort the data was disproportionately high. When sorting 1,000,000,000 integers with only 2 cores, the run time was nearly 20 minutes, or roughly four times that of the sequential algorithm. This is likely do to copying excessively large buffers between two cores that had a limited amount of memory allocated to them. It it likely that memory thrashing occurred as each core did not have enough memory allocated to it to contain all of the data at once. In cases where more than 16 cores where used, the large amount of memory required to store 1,000,000,000 integers was spread out over 3 to 4 cluster nodes. The averaged run time of the parallel algorithm are displayed in Table 3.

Parallel Bucket Sort Algorithm Run Time (s)

	Buckets										
Number of Integers		10	100	1000	10000	100000	1000000	10000000	100000000	500000000	1000000000
	2	3.11429E-05	4.97143E-05	0.000294333	0.00247583	0.0153635	0.195763	3.95827	56.3964	367.942	1209.15
	3	3.24286E-05	3.71429E-05	0.0002565	0.00158017	0.0120337	0.147396	2.75419	43.1032	268.971	757.481
	4	4.52857E-05	4.7E-05	0.000137667	0.00106667	0.00881533	0.107179	2.12378	36.1463	210.947	746.164
	5	5.68571E-05	6.05E-05	0.000132667	0.00109933	0.00721983	0.0999752	1.65481	25.8125	173.909	608.315
	8	0.00558743	0.00269033	0.00493617	0.00157433	0.00485667	0.0575872	1.05363	20.232	147.288	516.234
	9	0.00379771	0.00263433	0.00249467	0.002952	0.00586333	0.0426078	0.757756	15.7898	104.56	229.4
	16	0.00436529	0.0081665	0.00636617	0.00429733	0.006393	0.024805	0.506994	9.72685	77.1602	127.029
	17	0.00618971	0.008306	0.00443683	0.0067188	0.00705183	1.05015	0.523782	8.72081	63.2606	138.256
	24	0.0111226	0.0098115	0.0136157	0.00923833	0.010541	0.90056	1.13767	6.21615	56.0389	104.87
	25	0.00707686	0.0124312	0.00811067	0.0131748	0.00858533	1.60881	2.63322	6.23851	47.5391	108.119
	32	0.0149887	0.0255516	0.015077	0.0158005	0.0244642	1.48773	2.90242	4.34399	38.2476	83.6404
	33	0.011476	0.012494	0.0108768	0.0157308	0.0161787	1.60712	3.19697	4.58412	37.3944	93.864

Table 3: The averaged sorting time in seconds for the parallel algorithm across 2, 3, 4, 5, 8, 9, 16, 17, 24, 25, 32, and 33 cores. The number of integers sorted ranged from 10 to 1,000,000,000. With the exception of a small number of cores on a large number of integers, the parallel algorithm offered reasonable sort times.

In most cases, the parallel algorithm offered only a modest decrease in run time compared to the sequential algorithm. In some special cases, the run time was substantially worse. In other special cases, the decrease in run time was surprisingly high. The speed up of the parallel algorithm is show in Fig. 3.

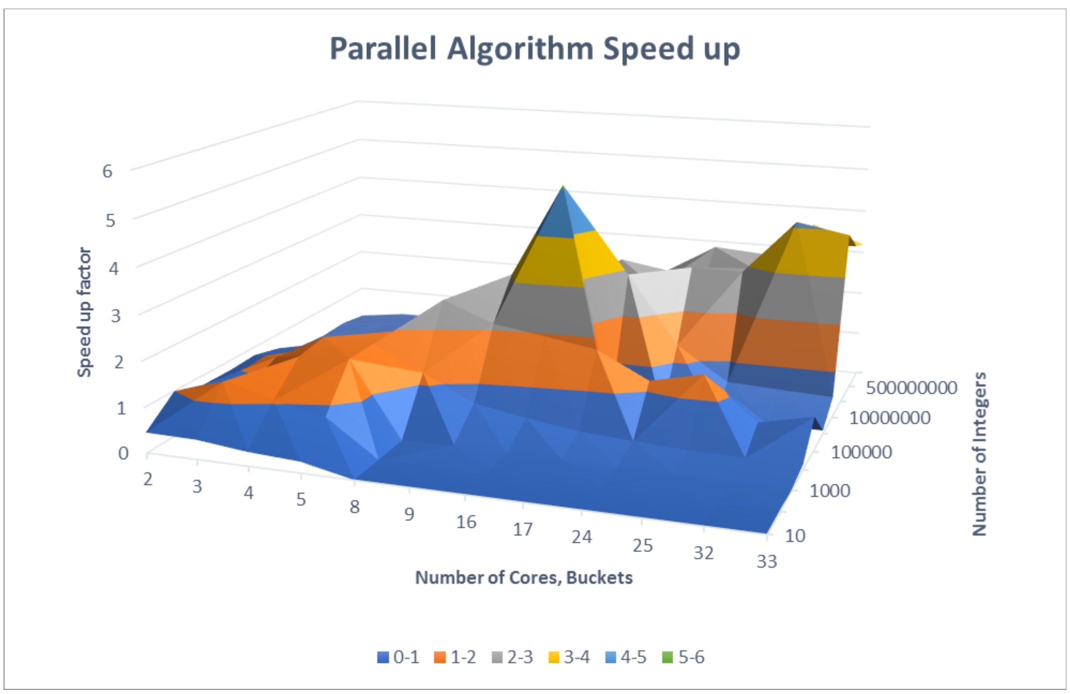

Figure 3: The speed up factor of the parallel algorithm. Although never exceptionally high, with the highest speed up around 5, the speed up factor is fairly consistent across a range of varying numbers of cores and integers being sorted.

The speed up of the parallel algorithm had a maximum of 5 when sorting 1,000,000 integers using 16 cores. The minimum speed up occurred with 32 cores sorting 10 integers, which is to be expected as a small amount of computations is being spread across a number of cores such that some processes have no work to do at all, but still have to communicate and synchronize with the working cores. This greatly increases the ratio of communication to computation. In the case of the maximum speed up, the relatively high level of speed up can be attributed to a fairly effective division of labor that overcame the communication time between two cluster nodes. When looking at the speed up for 1,000,000 integers with 17 cores, the speed up is less than 1. In this case, the additional communication time required to pass data to another cluster node, outweighed the increased division of labor. Table 4 displays the speed up factor for the parallel algorithm.

Speed Up for the Parallel Bucket Sort

	Buckets										
Number of Integers		10	100	1000	10000	100000	1000000	10000000	100000000	500000000	1000000000
	2	0.44954	0.997701	0.784146	0.77267	0.809308	0.645755	0.3820482	0.31784121	0.34600562	0.247924575
	3	0.43172	1.335383	0.899805	1.21063	1.033248	0.857656	0.5490725	0.415864715	0.4733224	0.395756461
	4	0.30915	1.055319	1.676509	1.79343	1.410475	1.179475	0.7120559	0.495904145	0.60351652	0.401758863
	5	0.24623	0.819835	1.739694	1.74015	1.722174	1.264464	0.9138511	0.694434867	0.73204952	0.492800605
	8	0.00251	0.018436	0.046757	1.21512	2.560149	2.195193	1.4352761	0.885977659	0.86436098	0.580701775
	9	0.00369	0.018828	0.092517	0.64804	2.120604	2.966945	1.9956952	1.135232872	1.21757842	1.30679163
	16	0.00321	0.006074	0.036254	0.44516	1.944908	5.096352	2.9827769	1.842847376	1.64994388	2.359917814
	17	0.00226	0.005972	0.052019	0.28472	1.763202	0.120378	2.8871744	2.055439804	2.01246906	2.168282028
	24	0.00126	0.005055	0.016951	0.20707	1.179566	0.140374	1.3292519	2.88363376	2.27181476	2.858567751
	25	0.00198	0.00399	0.028456	0.1452	1.448261	0.078577	0.5742969	2.873298272	2.6780061	2.772667154
	32	0.00093	0.001941	0.015308	0.12107	0.508245	0.084972	0.5210307	4.126413735	3.32857486	3.584129201
	33	0.00122	0.00397	0.021219	0.12161	0.768529	0.078659	0.473026	3.910259766	3.40452046	3.193748402

Table 4: The speed factor for the parallel bucket sort across all tested configurations of number of cores and number of integers to sort.

The speed up factor, given by dividing the sequential run time by the parallel run time, is a good measure of how much speed was gained by creating a parallel version of an algorithm. Using 2 cores never offered any speed up based on the speed up values in Table 4. At best, when sorting 100 integers, using 2 cores offered a speed up of 0.997701, which is only approximately on par with the sequential algorithm. In other cases, the speed up was well worth it. For instance, when sorting 1,000,000,000 integers, using 32 cores produced a speed up of 3.58. In terms of time, this reduced the sorting time from the sequential run time of 299.778 seconds to 93.864 seconds. Simply looking at times and the speed up factor, however, does not take into account the impact made on the system by the parallel algorithm. The efficiency is an effective measure for this and is graphed in Fig. 4.

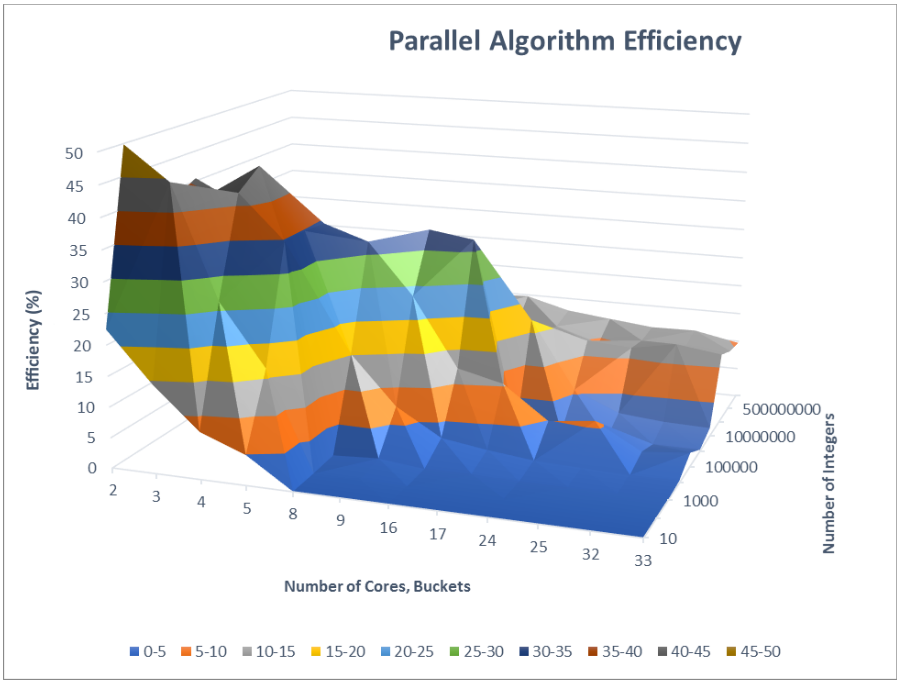

Figure 4: the parallel bucket sort efficiency as a percent.

In terms of efficiency, the parallel bucket sort never exceeded 50%. The highest efficiency of 49.885% was achieved when 2 cores were utilized to sort 100 integers. No speed up was achieved in this case even though the system used twice as many cores as the sequential algorithm. For numbers of cores above, the efficiency increases as the number of integers to sort increases. As more nodes in cluster are used, more computational work is needed to offset the communication cost. When only using cores on one box, 8 or less, the trend is that efficiency decreases as the number of integers increase. Table 5 has the efficiency for every tested combination of number of cores and number of integers tested.

Parallel Bucket Sort Efficiency (%)

	Buckets										
Number of Integers		10	100	1000	10000	100000	1000000	10000000	100000000	500000000	1E+09
	2	22.477033	49.885043	39.207292	38.633509	40.465389	32.287766	19.102411	15.89206	17.300281	12.396229
	3	14.390589	44.512769	29.993502	40.354308	34.44161	28.588519	18.302417	13.862157	15.777413	13.191882
	4	7.7287091	26.382979	41.912731	44.835797	35.261868	29.486886	17.801397	12.397604	15.087913	10.043972
	5	4.9246268	16.396694	34.793882	34.803016	34.44347	25.289272	18.277023	13.888697	14.64099	9.8560121
	8	0.0313203	0.230455	0.5844612	15.189001	32.001865	27.439909	17.940952	11.074721	10.804512	7.2587722
	9	0.0409604	0.2092035	1.0279694	7.2003914	23.562265	32.966056	22.174391	12.613699	13.528649	14.519907
	16	0.0200445	0.03796	0.2265884	2.7822508	12.155678	31.852197	18.642356	11.517796	10.312149	14.749486
	17	0.0133048	0.035127	0.3059948	1.6748439	10.371776	0.7081061	16.983379	12.090822	11.838053	12.7546
	24	0.0052446	0.0210637	0.0706292	0.8628002	4.9148563	0.5848907	5.5385496	12.015141	9.4658948	11.910699
	25	0.0079131	0.0159598	0.1138254	0.5808058	5.7930446	0.3143068	2.2971875	11.493193	10.712024	11.090669
	32	0.0029189	0.0060662	0.0478378	0.3783504	1.5882647	0.2655367	1.628221	12.895043	10.401796	11.200404
	33	0.0036968	0.01203	0.0643014	0.3685108	2.3288757	0.2383616	1.4334122	11.849272	10.316729	9.6780255

Table 5: the efficiency of parallel bucket sort as a percent.

Conclusion:

Significant, but limited speed up was achieved through the implementation of a parallel algorithm. On smaller sets of data, the sequential algorithm should be preferred as no speed up and even slow down typically occur. On larger sets of data, significant speed up was observed. In the case of using 32 cores to sort 1,000,000,000 integers, a significant decrease in the time to sort the data was observed. It is important to not, however, that any speed up observed did come at the cost of using larger amounts of system resources. When using 32 cores to sort 1,000,000,000 integers, only an 11.2% efficiency was achieved. The author recommends that the parallel algorithm analyzed in this report should only be used on large sets of data where computation speed is more important than the efficient use of computer resources.