Report on Project 3

Parallelization of Sorting Algorithms

Bucket Sort and Insertion Sort

CS415

April 5, 2017

Evan Su

Version 2.0

Table of Contents

Overview:

Most sorting algorithms are based on sequential code. In other words, each instruction is executed in order. The speed of sorting depends greatly on the runtime of the algorithm and clock speed of the computer. To achieve greater speeds, parallelization of algorithms is used. The goal of the experiment is to demonstrate the speedup achieved by parallelization.

Test Methodology:

Two different programs are written to test the sorting time, sequential and parallel. The sequential code will be used as a controlled variable. The code will be tested at various sizes starting at 100 unsorted items. For more detail of the sequential and parallel code, please see below.

Sequential

The sequential code first makes the buckets. Then, the buckets are filled with numbers within a certain range. Then, the bucket is sorted using insertion sort. When all the buckets are sorted, the data is collected in to a single array which can be displayed. The time starts when the sorting starts and ends when the sorting stops.

Parallel

The parallel code has the master send each slave portion of unsorted numbers. Then all the processors put the numbers in small buckets. Then, each of the small buckets placed in large buckets with matching ranges. Each large bucket is given to a processor which sort using insertion sort. When the sort is finished, the slaves send all the data back to the master. The master puts all the data into an array. The time starts when the little buckets begin to filled and ends when the all the cores finish sorting their buckets.

Data Analysis:

The raw data from the tests can be found in the file project3Analysis.

Inconsistencies in Sort Time.

The sort time for a single bucket in the sequential code is significantly different from the sort time of the parallel code. The difference in sequential time and parallel time can be found in Table 4.1. Ideally, the sort times of each bucket should be the same as the size and arrangement of the data is same in both methods. I believe the issue is cause by how the algorithm handles different bucket structure as the structure of the sequential bucket is not exactly the same as parallel bucket. In order the compensate for this difference, the time for sequential code is scaled down in the analysis of speedup and efficiency. The scale factor is calculated by the ratio of the sequential time over the parallel time plus a margin of error of 0.2.

Sequential

The run time of the sequential code exhibits a runtime of O(m*(n/m)^2) as show in graph 1.1. The runtime can be simplified to O(n^2 / m). The n^2 runtime is shown as the list size increases. The 1/m runtime effect is shown as the number of buckets increases.

Parallel

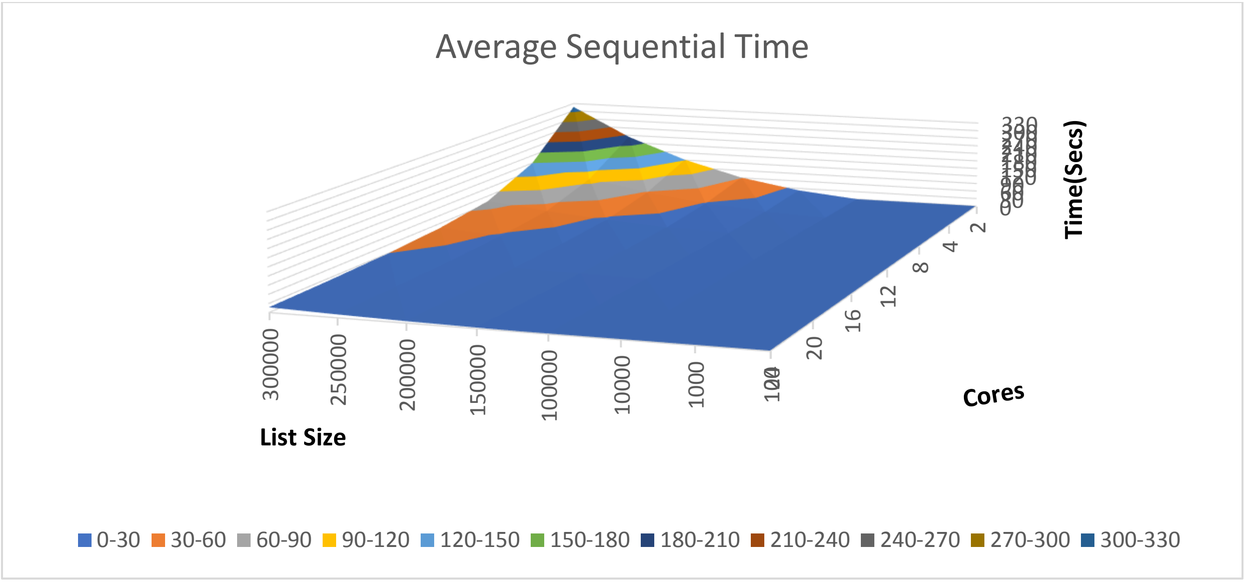
The run time of the parallel code is similar to the sequential code. As the size of the list increases, the run time increases exponentially. As the number of buckets increase, the run time decreases linearly. The trend can be seen in Graph 1.2. The maximum run time for the parallel code is around 125 seconds with 2 cores at a list size of 300,000. The other runtimes can be found in Table 1.2. The maximum speed up of parallelization is approximately 6.5 with 16 cores. Around the 20+ cores, the speed up drop significantly. The speedup at that point is similar to the speed up at four cores. The cause is due to too much communication time compared to computation time. Similarly, parallel sorting of small list produces significant slowdown.

Conclusion:

Parallelization of sorting algorithms can result in faster execution time but only at large numbers. For smaller numbers, sequential sorting would be more appropriate. For large numbers, increasing the number of cores would decrease the execution time but after a certain number of cores, the efficiency of each core will begin to drop. The tests have shown that speedup will occur with bucket sort and insertion sort. Future experiments on other sorting algorithms is necessary to determine the which is the most optimal in certain cases.

Graphs and Tables

Graph 1.1



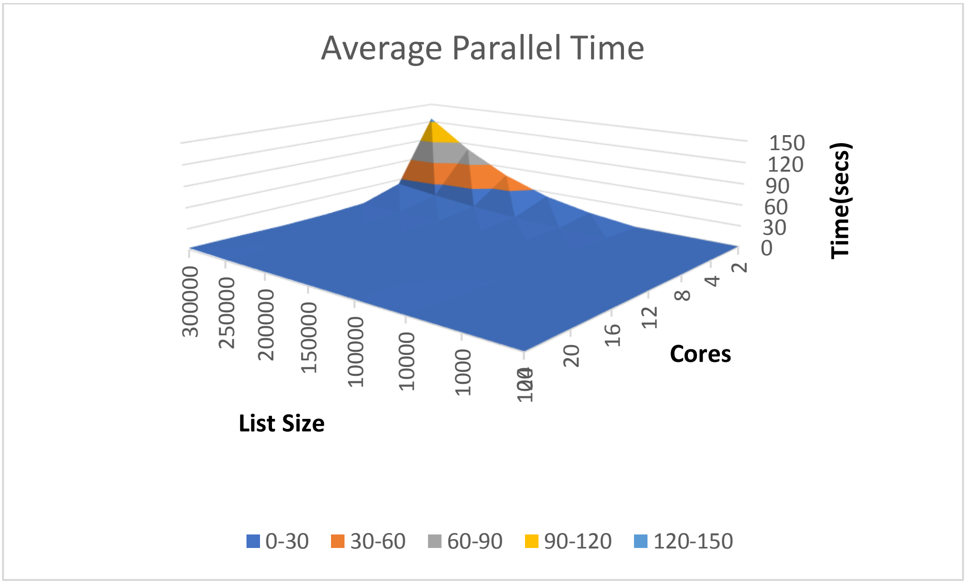
The Sequential time of bucket sort. The graph demonstrates the runtime of bucket sort is O(m*(n/m)^2). The time of the graph deceases linearly as the number of buckets used increases. The time of the graph increases exponentially as the list size increases.

Table 1.1

	Average Seq Time							
		Bucket						
		2	4	8	12	16	20	24
Size	100	0.00003	0.00002	0.00003	0.00002	0.00002	0.00002	0.00002
	1000	0.00102	0.00061	0.00042	0.00035	0.00032	0.00030	0.00028
	10000	0.07450	0.03225	0.01392	0.00994	0.00831	0.01027	0.00516
	100000	22.00072	9.63766	3.71475	2.22575	1.59802	1.22756	0.97675
	150000	60.97311	28.51068	11.99866	6.83139	4.78427	3.66361	2.95486
	200000	121.51912	58.17838	25.83029	15.09854	10.25381	7.77419	6.17342
	250000	203.47498	98.51819	45.25240	27.22416	18.54953	13.68567	10.83220
	300000	316.09522	149.69659	70.16982	43.13595	29.83641	21.96376	17.22645

The table shows the runtime of sequential sort. The execution time increased exponentially as size increases. The execution time decreases linearly with the number of cores.

Graph 1.2



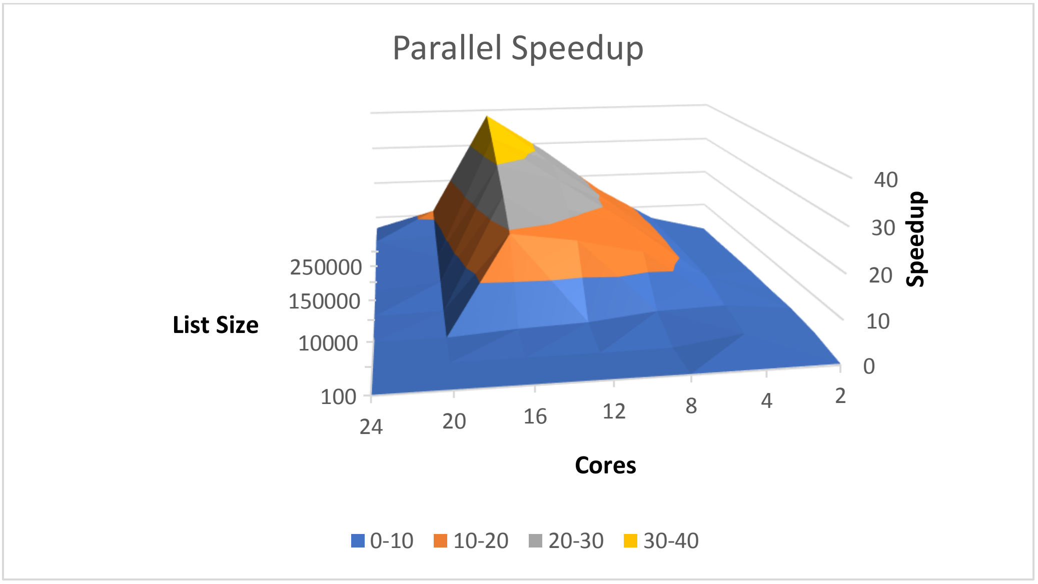
The parallel time of bucket sort. The graph demonstrates the runtime of bucket sort is O(m*(n/m)^2). The time of the graph deceases linearly as the number of buckets used increases. The time of the graph increases exponentially as the list size increases.

Table 1.2

	Average Parallel Time							
		Bucket						
		2	4	8	12	16	20	24
Size	100	0.0001	0.0001	0.0001	0.0080	0.0079	0.0030	0.0035
	1000	0.0006	0.0002	0.0016	0.0036	0.0058	0.0288	0.0147
	10000	0.0297	0.0079	0.0039	0.0055	0.0070	0.2076	0.1488
	100000	8.7912	1.5171	0.2742	0.1303	0.0822	0.9981	1.1180
	150000	22.7685	4.3899	0.7193	0.3238	0.1892	0.9451	1.3142
	200000	46.1926	8.8572	1.5026	0.5913	0.3436	1.3012	1.3704
	250000	80.6510	15.0811	2.7371	0.9946	0.5394	1.7509	1.5892
	300000	125.2459	23.0610	4.4266	1.5440	0.7761	1.9509	2.5177

The table shows the runtime of parallel sort. The execution time increased exponentially as size increases. The execution time decreases linearly with the number of cores.

Graph 2.1



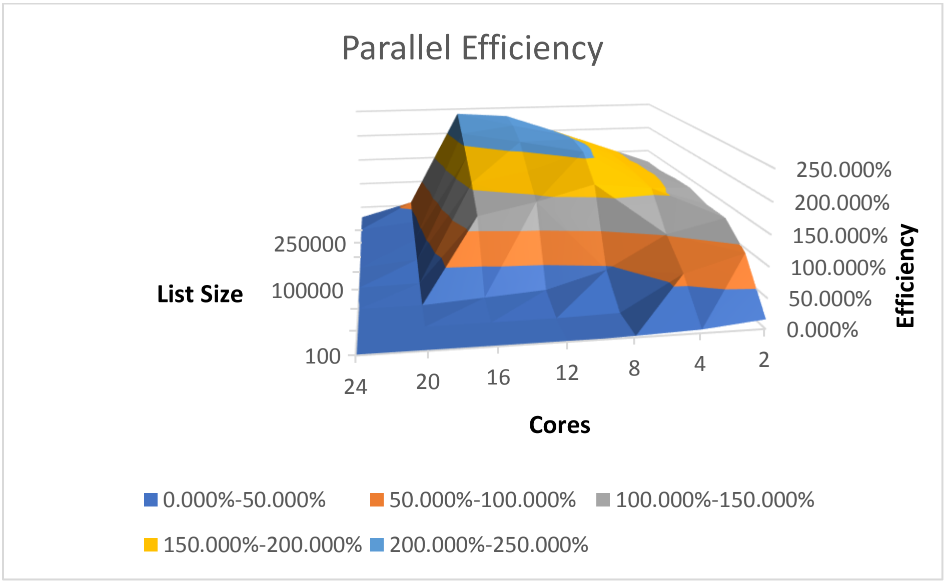
The speedup of parallel time reaches a peak at 16 cores. After 16 cores, the speedup drops significantly as another box is introduced to the calculations. The super-linear speedup is due to an algorithm issue where the sort time of sequential and parallel for a single bucket was different. Please see Graph 3.1 for data compensated for this issue.

Table 2.1

	Speedup							
		bucket						
		2	4	8	12	16	20	24
Size	100	0.3137255	0.2354949	0.19685	0.003033	0.002626	0.007892	0.006748
	1000	1.84843	2.5304709	0.26651	0.096997	0.054721	0.010315	0.019196
	10000	2.5062235	4.0643928	3.56197	1.792293	1.190883	0.049467	0.034704
	100000	2.5025874	6.3527998	13.5498	17.08649	19.44606	1.229834	0.873641
	150000	2.6779627	6.4946202	16.6812	21.10081	25.28607	3.876319	2.248357
	200000	2.630706	6.5684563	17.1902	25.53412	29.8406	5.974416	4.504742
	250000	2.5229072	6.5325434	16.5328	27.37281	34.39002	7.81634	6.815939
	300000	2.5237969	6.4913264	15.8518	27.93743	38.44347	11.25803	6.842258

The speedup of parallel time reaches a peak at 16 cores with a speed of 19. After 16 cores, the speedup drops significantly as another box is introduced to the calculations. The super-linear speedup is due to an algorithm issue where the sort time of sequential and parallel for a single bucket was different. Please see Table 3.1 for data compensated for this issue.

Graph 2.2



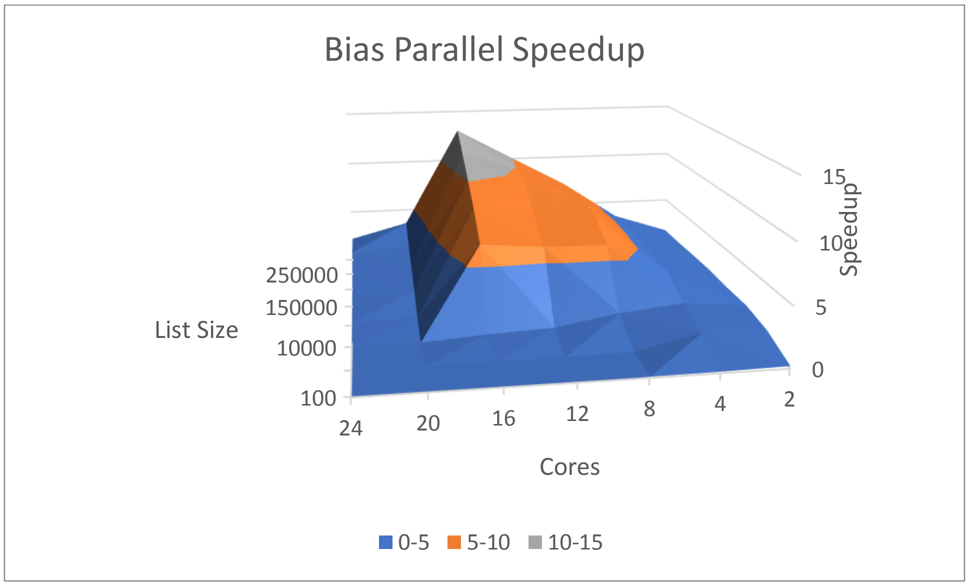
The efficiency increases as the list size increases. The efficiency also increases as the number of cores increases up to a certain point. At 20 cores, the efficiency drops significantly. The super-linear speedup is due to an algorithm issue where the sort time of sequential and parallel for a single bucket was different. Please see Graph 3.2 for data compensated for this issue.

Table 2.2

	Efficiency							
		bucket						
		2	4	8	12	16	20	24
Size	100	15.686%	5.887%	2.461%	0.025%	0.016%	0.039%	0.028%
	1000	92.421%	63.262%	3.331%	0.808%	0.342%	0.052%	0.080%
	10000	125.311%	101.610%	44.525%	14.936%	7.443%	0.247%	0.145%
	100000	125.129%	158.820%	169.372%	142.387%	121.538%	6.149%	3.640%
	150000	133.898%	162.366%	208.516%	175.840%	158.038%	19.382%	9.368%
	200000	131.535%	164.211%	214.878%	212.784%	186.504%	29.872%	18.770%
	250000	126.145%	163.314%	206.660%	228.107%	214.938%	39.082%	28.400%
	300000	126.190%	162.283%	198.147%	232.812%	240.272%	56.290%	28.509%

The efficiency increases as the list size increases. The efficiency also increases as the number of cores increases up to a certain point. Efficiency appears to peak at 16 cores with a list size of 300,000. At 20 cores, the efficiency drops significantly. The super-linear speedup is due to an algorithm issue where the sort time of sequential and parallel for a single bucket was different. Please see Table 3.2 for data compensated for this issue.

Graph 3.1



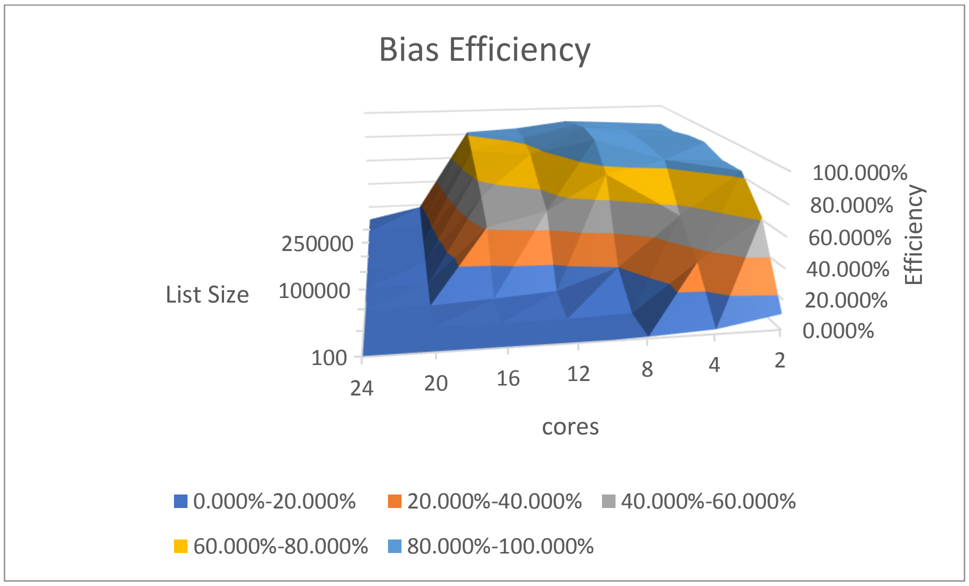
The speedup of parallel time reaches a peak at 16 cores. After 16 cores, the speedup drops significantly as another box is introduced to the calculations.

Table 3.1

	Bias speedup							
		bucket						
		2	4	8	12	16	20	24
size	100	0.209	0.126	0.088	0.001	0.001	0.003	0.002
	1000	1.234	1.354	0.120	0.035	0.019	0.003	0.006
	10000	1.673	2.174	1.598	0.646	0.405	0.016	0.011
	100000	1.671	3.398	6.077	6.154	6.608	0.396	0.279
	150000	1.788	3.474	7.482	7.600	8.592	1.247	0.717
	200000	1.756	3.513	7.710	9.197	10.140	1.922	1.437
	250000	1.684	3.494	7.415	9.859	11.686	2.514	2.174
	300000	1.685	3.472	7.110	10.063	13.063	3.621	2.182

The speedup of parallel time reaches a peak at 16 cores with a speed of 19. After 16 cores, the speedup drops significantly as another box is introduced to the calculations.

Graph 3.2



The efficiency increases as the list size increases. The efficiency constantly high at high list sizes and cores less than or equal to 16. At 8, 12, and 20 cores, the efficiency drops significantly due to high communications time.

Table 3.2

	Bias Efficiency							
		bucket						
		2	4	8	12	16	20	24
size	100	10.471%	3.149%	1.104%	0.009%	0.006%	0.013%	0.009%
	1000	61.694%	33.838%	1.494%	0.291%	0.116%	0.017%	0.026%
	10000	83.649%	54.349%	19.969%	5.380%	2.529%	0.080%	0.046%
	100000	83.527%	84.950%	75.963%	51.287%	41.299%	1.978%	1.161%
	150000	89.381%	86.846%	93.519%	63.336%	53.702%	6.234%	2.988%
	200000	87.803%	87.834%	96.372%	76.643%	63.374%	9.608%	5.987%
	250000	84.206%	87.353%	92.687%	82.162%	73.036%	12.571%	9.058%
	300000	84.235%	86.802%	88.869%	83.857%	81.645%	18.106%	9.093%

The efficiency increases as the list size increases. The efficiency constantly high at high list sizes and cores less than or equal to 16. At 8, 12, and 20 cores

Table 4.1

Sequential vs Parallel average one bucket calculations at 300,000 list size							
Bucket	2	4	8	12	16	20	24
Sequential	194686234	37424147	8771227	3594662	1864776	1098188	717769
Parallel	162238528	22415413	4321523	1395284	679859	377518	244540

The table demonstrates the algorithm discontinuities that occurred with sorting the same buckets. The times for sorting a single bucket should be the same as both parallel and sequential use the same algorithm. However, tests show that the sorting times are different. I believe the structure is the cause of the execution difference