Tyler DeFoor

PA3 - Bucketsort

3/29/2017

Dr. Harris

Sequential

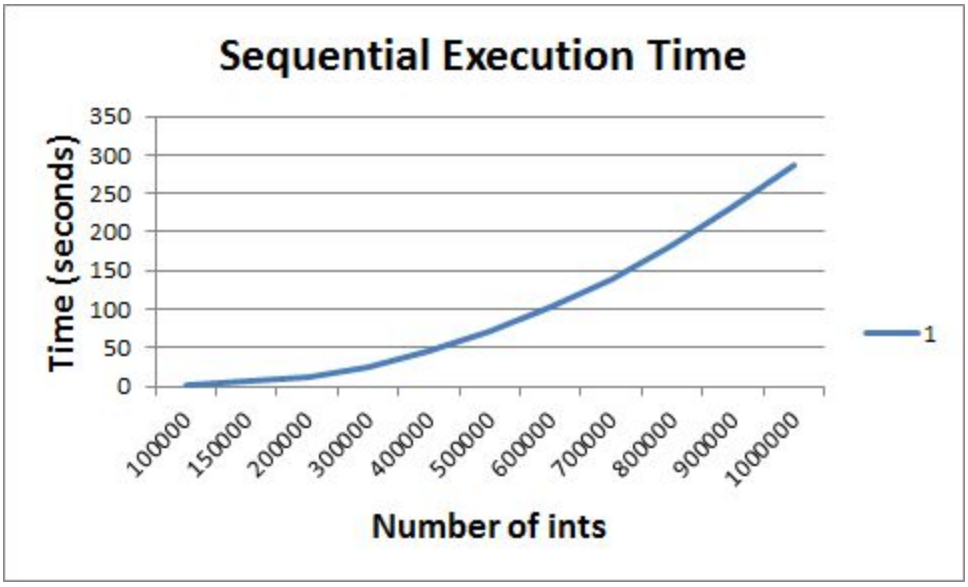
Sequential Implementation

This implementation of Sequential Bucketsort takes in either a file or generates an amount of random numbers, then sorts them using a bucket sorting technique. After acquiring the numbers, it puts them all in buckets based on the range in which they fall. Currently, it only sorts the numbers into ten buckets regardless of range.

After the numbers are in their buckets, the program bubble sorts the buckets and puts them back into an overall vector. This vector is sorted, as the buckets themselves are sorted.

Bubblesort is an $O(N^2)$ algorithm, and the graph below shows it rather well. It took until just over one million ints for it to reach 5 minutes of sorting time. The timer was started when the ints where in small buckets ready to be sorted, and it ended when all of the small buckets were sorted. The tabular data is in the **Dynamic Table** section of the report.

Sequential Graph



Dynamic

Dynamic Implementation

This implementation of a parallelized Bucketsort takes in either a file or generates an amount of numbers, then sorts them using a bucket sorting technique. Let K be the number of ints to be sorted and N be the number of processors, including the master process. The master process reads in K/N numbers, then sends it to the first slave. It does this until the last K/N numbers, which it takes care of. All of the processes then put their numbers into small buckets based on the range in which they fall.

After they are put into their buckets, each processor bubble sorts them. The time is started when each processor has its data, and it is ended after each processor sorts its small bucket.

Analysis

As bubblesort is an $O(N^2)$, the more cores that worked on the overall set of data it got increasingly faster and more efficient. On 10 cores there was commonly greater than 10 speedup, while 16 cores was averaging out at around 25 times speedup.

The superlinear speedup is due to the nature of bubblesort. Let K be equal to one million. As N increases, K/N gets smaller and smaller. For instance with 2 processors, each has to deal with 500,000 integers. With twenty processors, each has to deal with 50,000 integers. That is a large difference for an $O(N^2)$ algorithm, as $50{,}000^2$ is 2,500,000,000 and $500{,}000^2$ is 250,000,000,000.

With the superlinear speedup, there was also commonly sublinear speedup on 2 cores. On top of that the efficiency doesn't pass 70% until 8 cores. This is in line with an $O(N^2)$ algorithm as inefficient as bubblesort is. The fewer numbers there are, the more work fewer cores have to do. The work an individual processor has to do goes down exponentially as more processors are added to the problem.

Execution Time Graph and Table

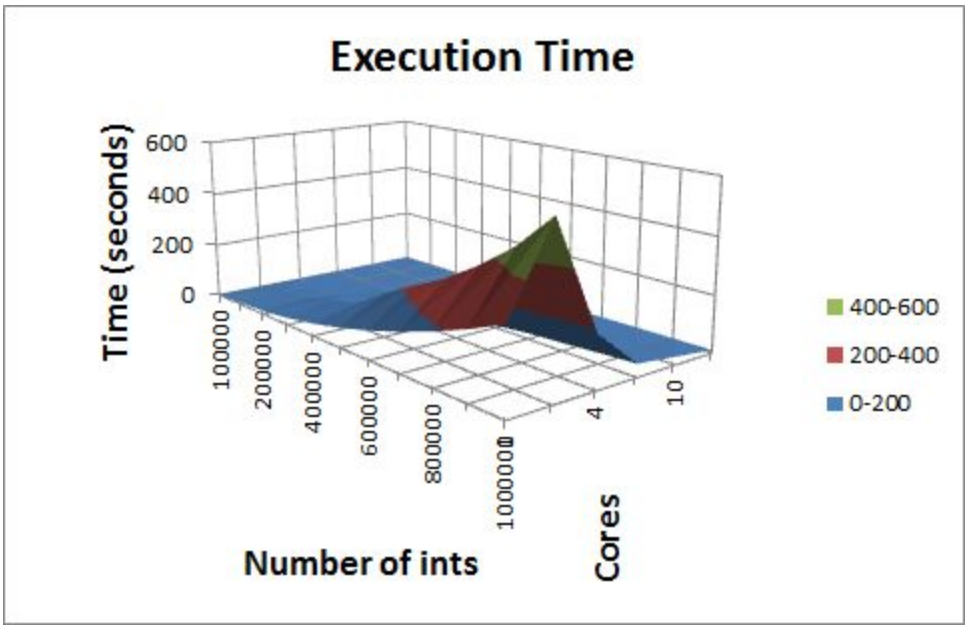


Table 1: Execution Time

# Ints Cores	100000	150000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
1	2.856	6.411	11.86	25.760	45.76	71.551	103.0	140.2	183.2	231.8	287.9
2	1.361	16.15	28.64	57.245	80.28	157.11	220.5	280.5	358.6	450.8	572.6
4	1.817	4.066	7.457	16.836	29.18	42.574	70.48	93.33	113.78	147.6	183.4
8	0.447	1.007	1.874	4.1886	8.555	12.232	16.85	21.09	28.77	37.95	45.95
10	0.284	0.641	1.174	2.5255	4.983	7.8329	11.24	13.56	16.98	21.09	28.46
16	0.111	0.252	0.446	1.0222	2.012	2.5884	4.451	5.766	7.542	9.013	11.43

Speedup Graph and Table

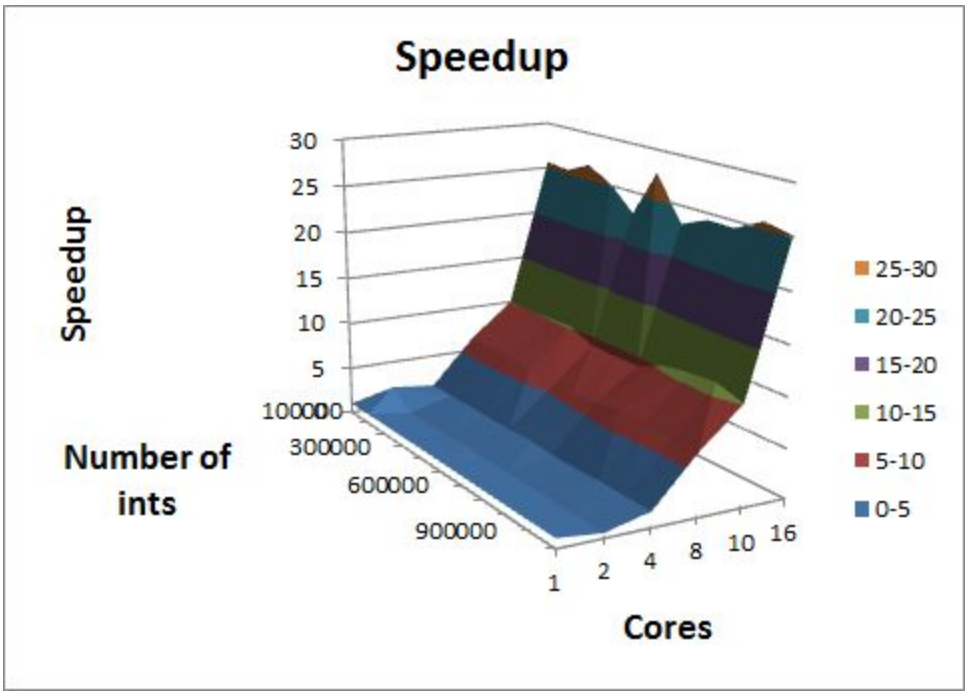


Table 2: Speedup

# Ints Cores	100000	150000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
1	1	1	1	1	1	1	1	1	1	1	1
2	2.0977	0.3969	0.4141	0.45	0.57	0.4554	0.4674	0.5	0.51098	0.5142	0.502856
4	1.5713	1.5767	1.5908	1.53	1.5681	1.6806	1.4626	1.5027	1.61080	1.57	1.57
8	6.3795	6.3640	6.3303	6.1500	5.3494	5.8492	6.1154	6.6490	6.37000	6.1093	6.266881
10	10.047	9.9953	10.100	10.200	9.1841	9.1346	9.1675	10.342	10.7920	10.989	10.11591
16	25.662	25.370	26.550	25.200	22.741	27.642	23.159	24.324	24.3018	25.724	25.1949

Efficiency Graph and Table

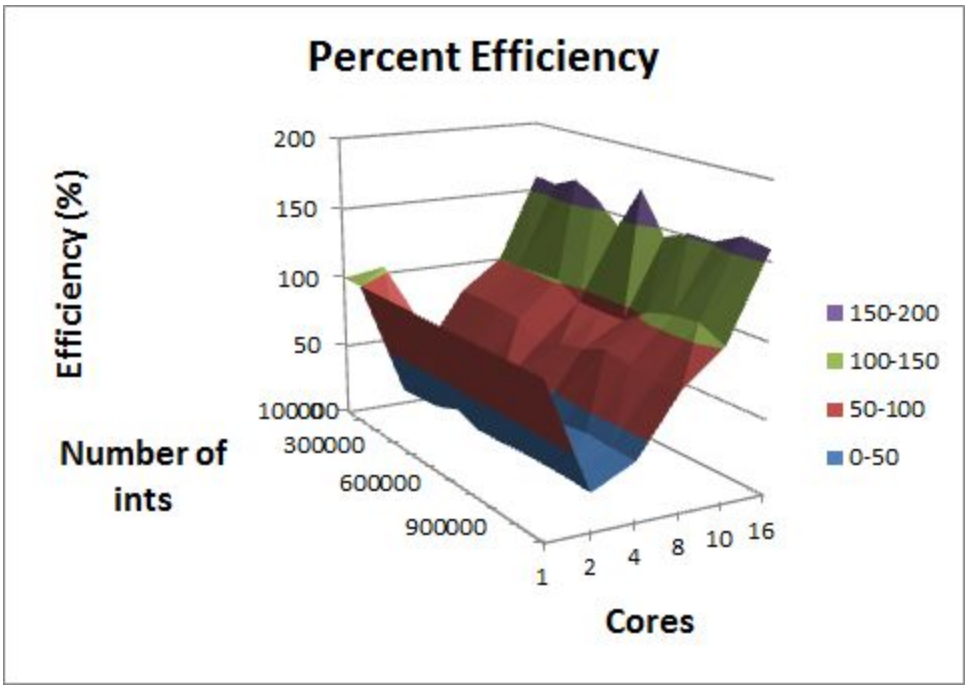


Table 3: Percent Efficiency

# Ints Cores	100000	150000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
1	100	100	100	100	100	100	100	100	100	100	100
2	104.88	19.846	20.707	22.5	28.5	22.770	23.371	25	25.5492	25.712	25.14282
4	39.284	39.419	39.770	38.250	39.203	42.015	36.565	37.567	40.2700	39.25	39.25
8	79.744	79.550	79.129	76.875	66.868	73.115	76.442	83.113	79.6250	76.367	78.33602
10	100.47	99.953	101.00	102.00	91.841	91.346	91.675	103.42	107.920	109.89	101.1591
16	160.38	158.56	165.94	157.50	142.13	172.76	144.74	152.03	151.886	160.77	157.4681

Future Work

Bucketsort is a very versatile algorithm that can be done many ways. The biggest change that I would like to analyze is the effect of other sorting algorithms on the speedup. In class, many people mentioned that C++ standard sort gave the lowest speedup because of how optimized it is. Implementing Bucketsort with quicksort, insertion sort, or even a recursive or parallelized mergesort will give different data than my bubblesort implementation. Likewise, getting to use all 96 nodes on the cluster would be a fun task that I would like to carry out when it is not so active.