Jake Wheeler

CS415

04/07/2017

PA3 - Bucket Sort

Overview

This project focuses on the sorting algorithm “Bucket sort”. Bucket sort is one of the most parallelizable sorting algorithms out there, and scales very well when parallelized. Bucket sort relies on splitting up a data set to sort, placing each piece of data in a smaller ‘bucket’, running some sorting algorithm on each bucket, and reconstructing the data set from the sorted buckets. This can be easily parallelized due to the allocation of buckets, each processor can simply be its own bucket.

A bucket sort program was written both sequentially and parallel, in order to compare execution time between the two methods. The programs generate an array of random integers of size specified by the user, and proceed with the bucket sort algorithm. The user also may specify a seed for the random number generation, to ensure the same set of numbers will be used over various tests.

Sequential

In writing and testing the sequential implementation of bucket sort, I initially found that 500 million integers was a good testing point but later after optimizations was able to go to around 2 billion integers. In my sequential bucket sort, 2 billion integers took around 2 minutes to sort, and anything above 2 billion integers started thrashing on my local machine. The execution time is surprisingly linear when increasing the number of integers, which can be seen in Figure 1, below.

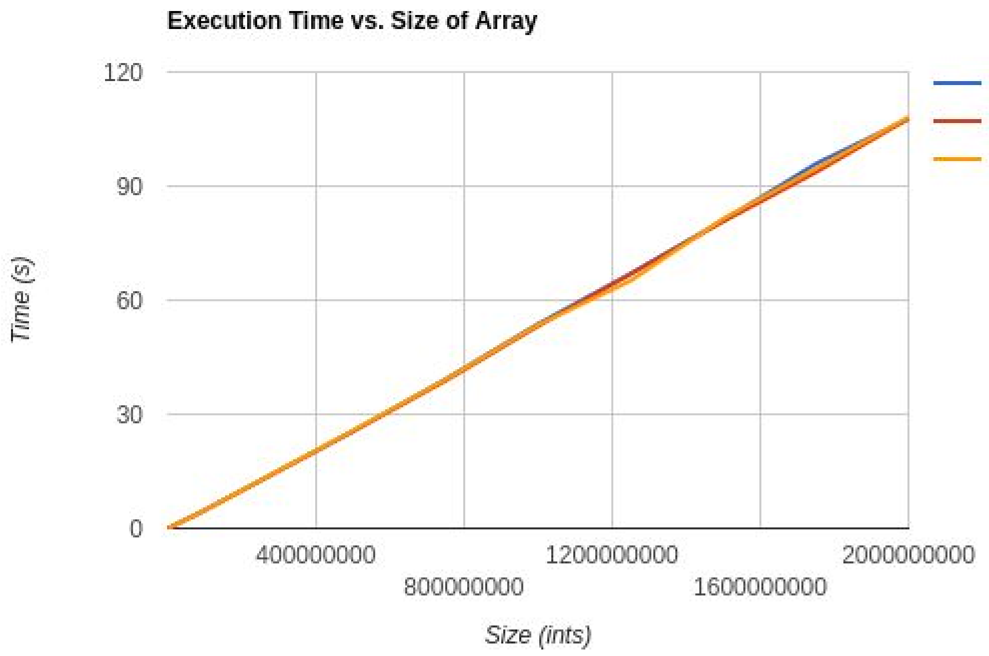

Figure 1: Figure 1, above, shows the relationship between the number of integers being sorted and the execution time when running bucket sort sequentially.

Size (ints)	1m	10m	100m	500m	750m	1b	1.25b	1.5b	1.75b	2b
Test 1 Time (s)	0.040052	0.432654	4.7631	25.5254	39.2026	53.7443	67.0068	81.0053	95.8644	107.671
Test 2 Time (s)	0.040037	0.431894	4.75813	25.5168	38.8104	53.1961	66.8412	80.716	93.646	107.733
Test 3 Time (s)	0.040052	0.431795	4.76037	25.7214	39.1344	53.5173	65.134	81.435	94.5324	108.123

Table 1: Table 2, above, shows data for various tests logging the execution time of sequential bucket sort given different quantities of integers.

Parallel

Bucket sort was fairly easy to parallelize, given the structure of the algorithm. The timing was done only on the actual sorting, that is, after the master node has assigned all of the proper numbers to the proper processors the timer began. The timer then ends after all of the processors are finished with their sorting. First, the numbers are generated and distributed to the proper process (or bucket), the timer starts, the processes sort their data, and the timer ends. For my tests, bucket sort was run with 2, 3, and 4 cores on my local machine due to high traffic on the h1 cluster. Although it is run with a small amount of cores, significant trends and speedup can be seen below in Figures 2/3 and Tables 2/3, below. NOTE: Because the tests for parallel were run on a different machine than the sequential portion of the report, sequential tests were rerun to be consistent with parallel data.

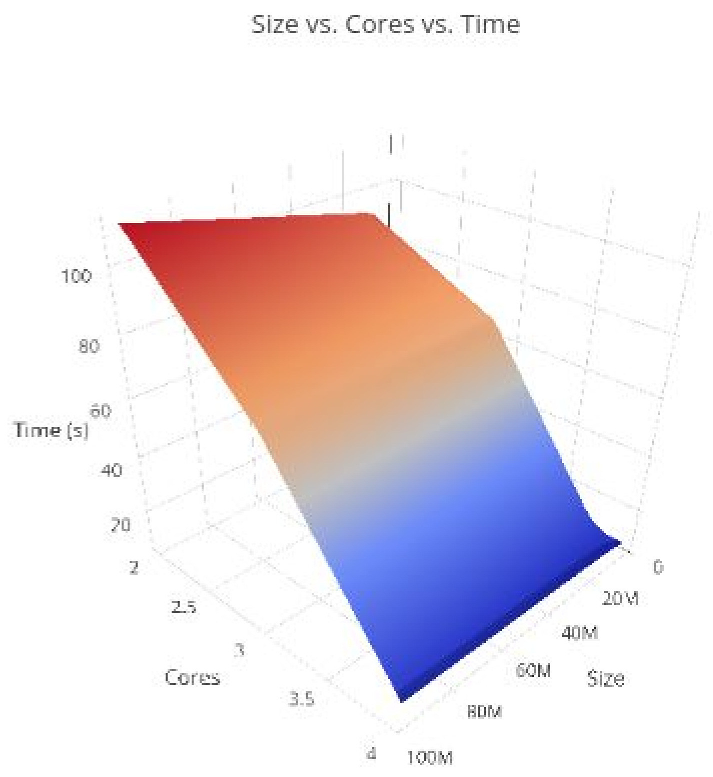

Figure 2: Figure 2, above shows a 3D surface graph of number of integers (x) vs. number of cores (y), and execution time (z).

	1000000 ints	10000000 ints	100000000 ints	500000000 ints	750000000 ints
2 cores	0.124991 s	1.05442 s	11.9582 s	125.331 s	169.723 s
3 cores	0.120789 s	0.984599 s	10.5138 s	70.6879 s	112.169 s
4 cores	0.096514 s	0.685851 s	7.38305 s	71.8032 s	92.6656 s

Table 2: Table 2, above shows data pertaining to Figure 2, showing the execution time for different tests with varying number of cores and sizes.

Regardless of the small number of cores these tests were run on, significant things can be seen. Slightly sublinear speedup was achieved when looking at this graph, as the graph trends upwards and to the left, in other words the time increases as the number of cores decrease and the size increases. The speedup of the parallelized program can be seen in Figure 3, below.

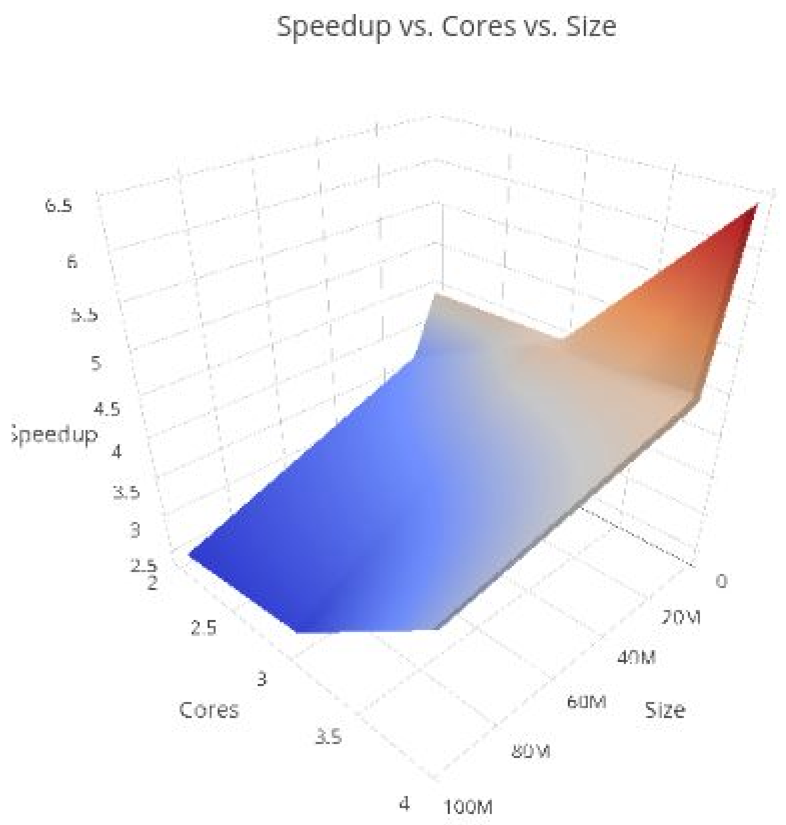

Figure 3: 3D Surface graph showing the relationship between speedup, number of cores, and number of integers. Accompanying data in Table 3.

	4 cores	3 cores	2 cores
750000000 ints	4.418198339	3.649983507	2.412254085
500000000 ints	4.454703969	4.524989425	2.552137939
100000000 ints	6.504168332	4.567387624	4.015704705
10000000 ints	4.06231091	2.829720526	2.642343658
1000000 ints	2.594680565	2.073226867	2.003528254

Table 3: Data recorded from finding speedup in relation to number of cores and size of input data.

In Figure and Table 3, the speedup can be seen to be fairly linear when increasing the number of cores. This trend is more closely linear when run on a higher number of input data, and less so on smaller data sets like the 1,000,000 integer test. Although when run on 2 cores the speedup is shown to be 2, when increasing the number of cores to 3 or 4 does not result in a significant speedup difference. This is likely due to the fact that computation time is very small for smaller data sets while communication time stays relatively the same no matter the size of the data set. This means that the more computation is done, the more “worth it” the parallelization becomes. There was a significant spike in speedup when the test was run with 100,000,000 integers, but I attribute that outlier to differences in the system at different run times. For example, when the 100,000,000 integer test was run, a significant amount more memory could have been freed up, meaning less time spent during memory allocations.