Andrew Frost
CS415
PA3 Report
03/29/2017

Overview:

In Programming Assignment 3 Milestone 1, a sequential bucket sort was implemented and tested on a large number of integers. In this case sets of 10 to 100,000,000 integers were randomly generated and sorted by the algorithm. All of the sorted integers fall between 0 and 999. The times were recorded only for the sorting portion of the program. File I/O is not accounted in the timing results. Five trials were conducted for each number of integers sorted. The results of these five trials were averaged to create the timing data discussed in this report. Not every number of integers between 10 and 100,000,000 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 n buckets where n is the number of integers to be sorted. 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 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.

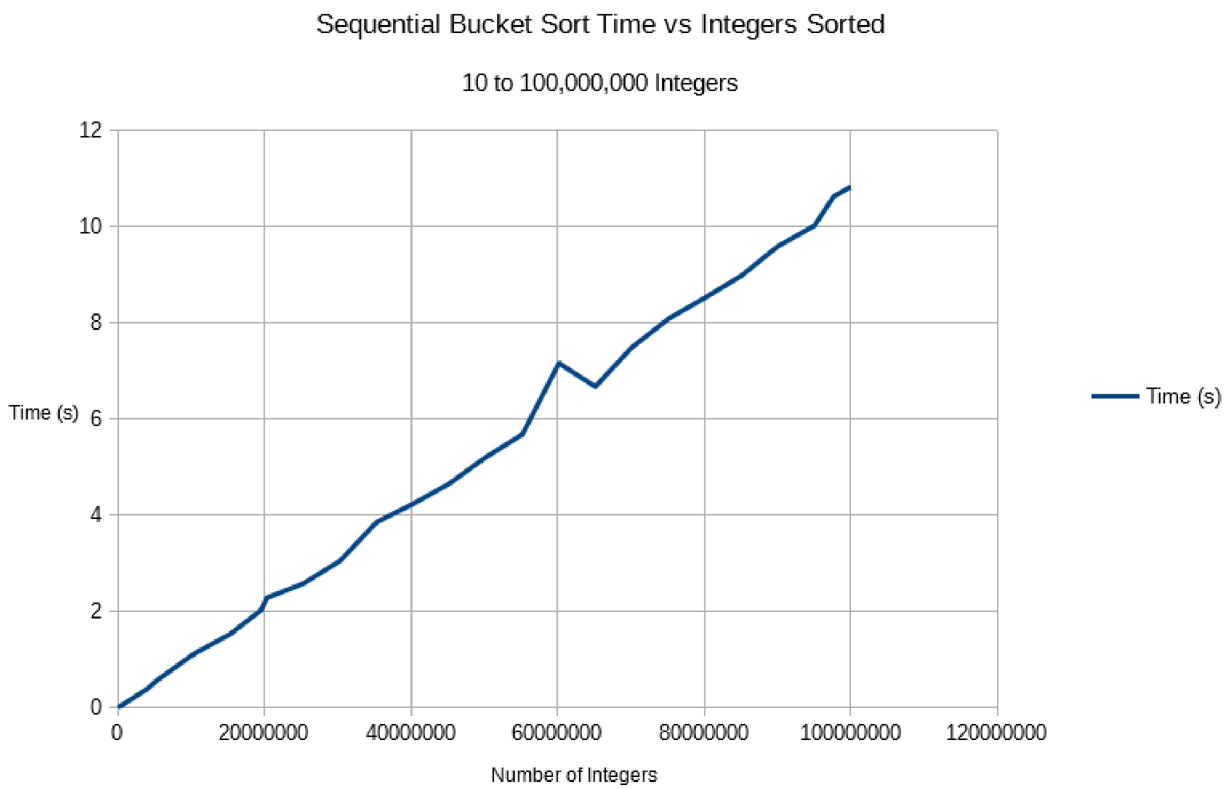

Figure 1: A graph of the averaged timing results of sorting 10 to 100,000,000 integers.

As the graph in Figure 1 shows, the time to sort an increasing number of integers increased almost linearly, which follows the O(n + k) computational run time of bucket sort. In the case of 60,000,000 integers, an abnormal jump in the time required to sort the integers occurs. This is likely caused by an excessive number of swaps in memory or poor partitioning by the sequential algorithm caused with this number of integers. The averaged timing results are displayed in Table 2.

The Averaged Timing Results of Sorting Integers

Number of Integers	Time (s)
10	1.06E-05
50	4.1E-05
250	0.0001168
1250	0.000474
6250	0.0020016
31250	0.0032006
156250	0.014023
400000	0.039554
781250	0.0744038
3906250	0.379205
5380000	0.571369
10360000	1.11791
15340000	1.5313
19531250	2.026566
20320000	2.283876
25300000	2.578568
30280000	3.047642
35260000	3.852568
40240000	4.23498
45220000	4.658368
50200000	5.207944
55180000	5.679766
60160000	7.1585
65140000	6.676182
70120000	7.488588
75100000	8.08504
80080000	8.520502
85060000	8.982212
90040000	9.59302
95020000	10.021496
97656250	10.62822
100000000	10.8297

Table 2: The average time to required to sort different numbers of randomly generated integers between 0 and 999. The number of integers to be sorted is over a range from 10 to 100,000,000 integers.

Overall, as the number of integers to be sorted increased so did the amount of time required to sort them. This was not the case when sorting 60,000,000 and 65,000,000 integers where the time briefly decreased. The author attributes this to inefficiencies when sorting 60,000,000 integers caused by either poor partitioning or excessive memory swapping.