PA3 – Bucket Sort
Documentation 1

Brian Conway

30 Mar. 2017

CS415

Bucket Sort - Sequential

Bucket sort involves taking an unsorted collection of numbers and partitioning them into their corresponding buckets, with their buckets being determined by the specific interval they fall into. Bucket sort requires the values in the collection to be uniformly distributed.

Methodology

The file generator.cpp uses c++11 random devices to generate uniformly distributed values from 0 to 999,999 to a file called data.txt. The number of values generated is determined by the command line argument passed to the executable. The first line of data.txt is the number of items in the file, followed by the randomly generated numbers.

The file sequential.cpp is responsible for reading in the numbers from the data file, bucket sorting them, and outputting statistics about the timing as well as optionally writing the sorted numbers to a file.

The program has one mandatory command line parameter for the file name of the input, and an optional parameter for if you want the sorted numbers output to a file. If the second command line parameter is “y”, then the results are output to the file “sorted.txt”.

After reading the number of items and all the individual numbers from the file, the program calls the bucketSort() function, which splits the data into the number of buckets specified by the NUM_BUCKETS constant, then uses std::sort() to sort each bucket. It times this using the Timer class, and it repeats this process the amount of times specified by the NUM_MEASUREMENTS constant. Afterwards, it calls the calcStatistics() function to get an average of the time taken to bucket sort the numbers and outputs it to the console.

Timing is done using the custom Timer class, which has start, stop, and resume functions similar to a stopwatch. Timer makes use of timevals and the gettimeofday() function in order to get an accurate reading of seconds and milliseconds at certain instants. When the elapsed time is to be given, the Timer takes the difference in seconds and milliseconds between when the timer was started and when the timer was stopped. It then combines the seconds and milliseconds reading into a double precision floating point variable.

Results

Below is a surface graph of the amount of numbers to sort, the number of buckets, and the time taken to sort those numbers with that amount of buckets. Data was taken for 4, 8, and 16 buckets with the amount of numbers to sort starting from 1,000 and ending at 10,000,000.

In general, it looks like the number of items to sort affected the time taken much more than the number of buckets used, with the biggest increase in time taken being from 1,000,000 to 10,000,000 items to sort. It does look like with more than a million numbers to sort, going from 8 to 16 buckets decreased runtime much more than when there were less than a million numbers to sort.

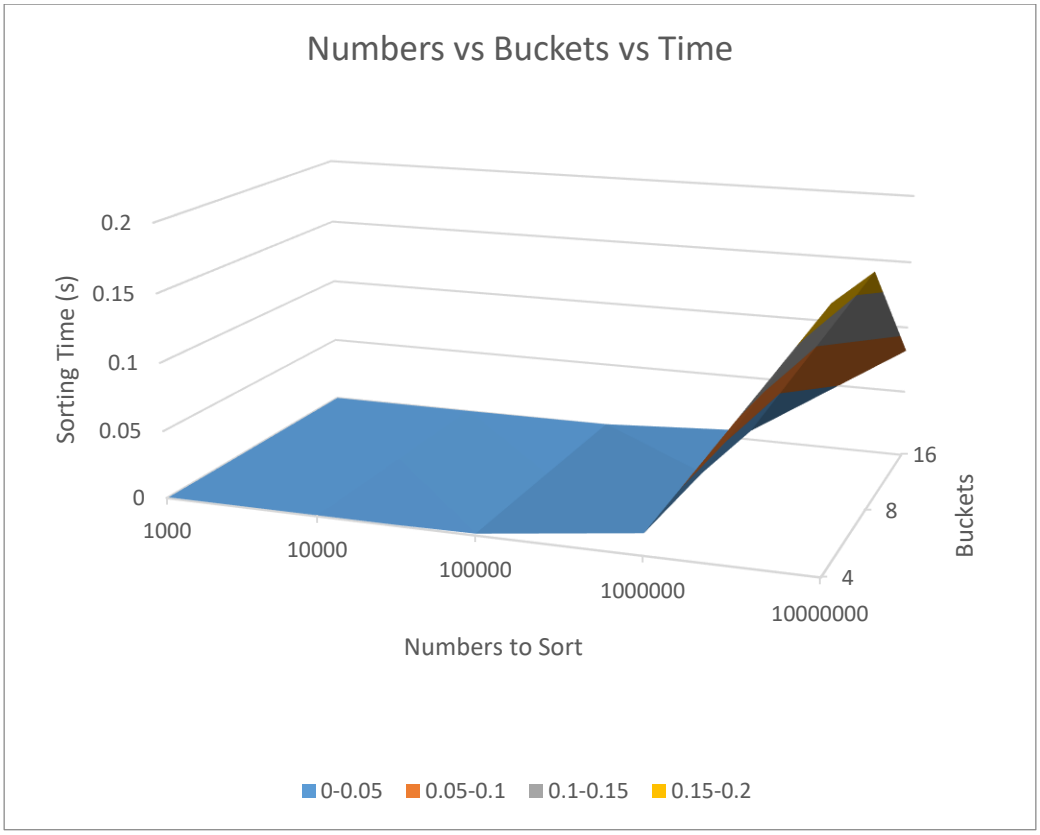

Fig. 1 Graph of runtimes with different amounts of numbers to sort and numbers of buckets used in the bucket sort.

Measurements

	1000	10000	100000	1000000	10000000
4	**1.86E-05**	**0.0001554**	**0.0013524**	**0.015875**	**0.180381**
8	**0.0000182**	**0.0001434**	**0.001261**	**0.015059**	**0.171701**
16	**0.0000165**	**0.0001702**	**0.0005912**	**0.007112**	**0.082027**

Table 1: Timings of sequential bucket sort calculations, with the top row being the number of numbers to sort and the leftmost column indicating the amount of buckets used.