PA3 – Bucket Sort Documentation

Brian Conway

6 Apr. 2017

CS415

Bucket Sort - Sequential

This sequential implementation of 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. After partitioning into the buckets, the buckets are sorted using bubble sort.

Bucket Sort - Parallel

This parallel implementation of bucket sort sort involves the master node taking an unsorted collection of numbers and partitioning them into their corresponding regions based on the number of processes specified. Starting from process 1 and with total number of items being *n*, process *k*'s region will be the first *n / k* numbers. The master will take the last region, and depending on input size and number of processes may have a slightly larger region than the rest. After distributing the regions, each process will put the numbers in their region into one of *n* buckets, with the buckets being determined by which interval the number falls into. Once the numbers in the region are in their proper buckets, each process will send bucket *k* to process *k*. When a process receives one of these buckets, they empty the contents into one big bucket. After all buckets are exchanged, each process sorts their big bucket with bubble sort.

Methodology

The file sequential.cpp is responsible for generating a specified number of random values, 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 number of items to generate, 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 "sortedSeq.txt".

The program first calls generateNumbers(), a function which c++11 random devices to generate uniformly distributed values from 0 to 999,999. The generator is seeded appropriately so that this sequential version generates the same numbers as the parallel one. Afterwards, the program calls the bucketSort() function, which splits the data into the number of buckets specified by the NUM_BUCKETS constant, then uses the bubbleSort() function 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.

The file parallel.cpp is responsible for generating a specified number of random values, bucket sorting them in parallel, 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 number of items to generate, 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 "sortedPar.txt".

The program's master process first calls generateNumbers(), a function which c++11 random devices to generate uniformly distributed values from 0 to 999,999. The generator is seeded appropriately so that this parallel version generates the same numbers as the sequential one.

Afterwards, the master node divides the numbers into their corresponding regions based on the number of processes specified. Starting from process 1 and with total number of items being *n*, process *k*'s region will be the first *n / k* numbers. The master will take the last region, and depending on input size and number of processes may have a slightly larger region than the rest. The master sends the region size as well as the numbers in each region to their corresponding process using MPI_Send(). After everyone has their regions, a barrier is called to ensure all processes are at the same spot. After the barrier, the master starts the timer and everyone takes their region and puts the numbers into one of *n* small buckets, with the buckets being determined by which interval the number falls into. Once the numbers in the region are in their proper buckets, each process will send bucket *k* to process *k*. When a process receives one of these buckets, they empty the contents into one big bucket. After all buckets are exchanged, each process sorts their big bucket with bubble sort. A barrier is called after this to ensure all processes have finished sorting.

Afterwards, the master stops the timer and 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 – Sequential

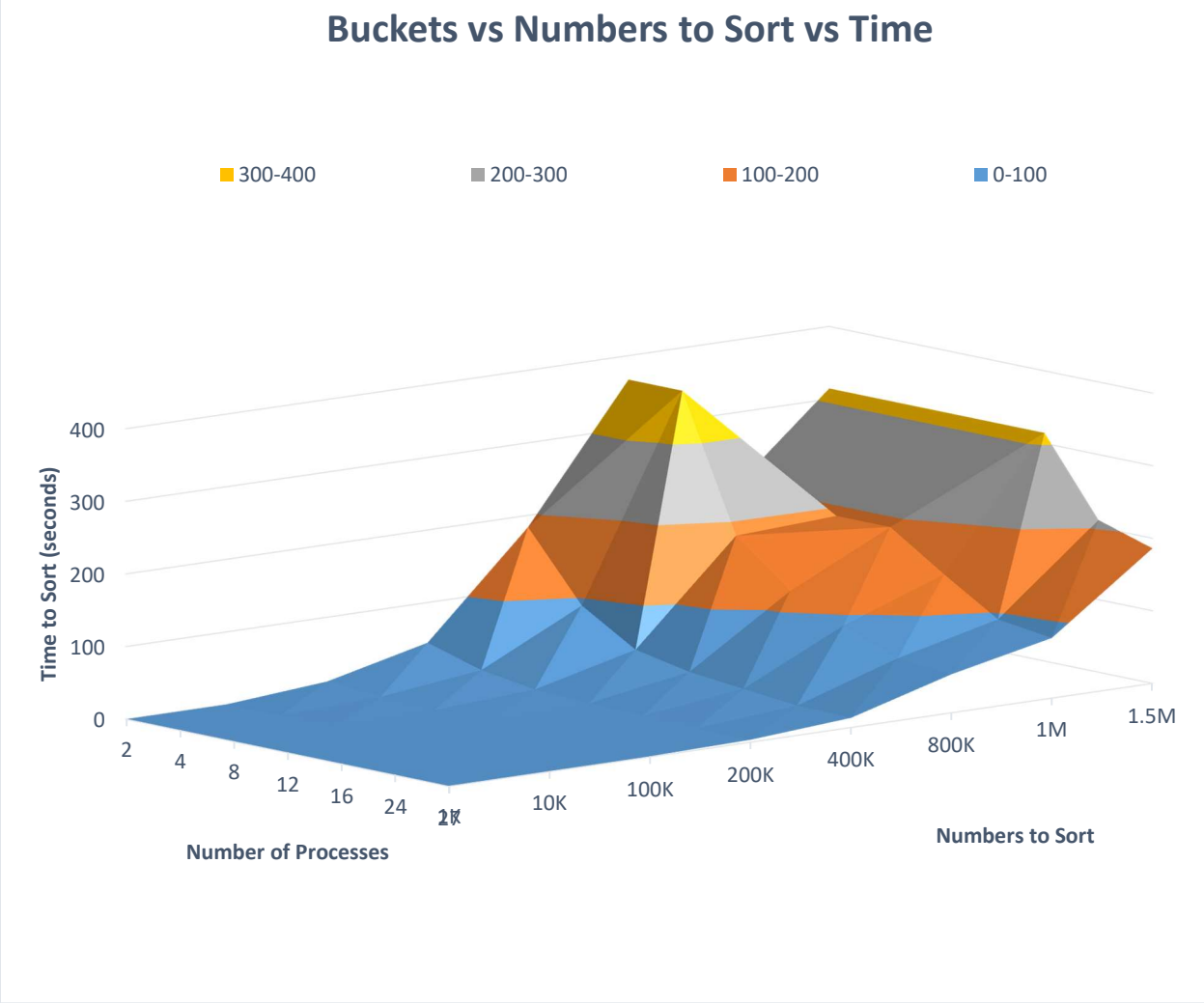

Fig. 1 Graph of sequential runtimes with different amounts of numbers to sort and numbers of buckets used in the sequential bucket sort.

	1K	10K	100K	200K	400K	800K	1M	1.5M
2	0.002569	0.118276	11.3933	44.6497	183.79	366.972	189.545	314.497
4	0.001775	0.060603	5.68567	22.3398	91.0343	366.972	189.545	314.497
8	0.000859	0.032045	2.81298	11.1515	45.548	182.509	189.545	314.497
12	0.000223	0.021233	1.88346	7.38929	30.3257	121.272	189.545	314.497
16	0.000126	0.018709	1.38915	5.54659	22.3736	89.4409	139.797	314.497
24	0.000156	0.010371	0.927758	3.6982	14.817	59.5779	93.0543	209.774
27	0.00026	0.012691	0.82578	3.28751	13.1505	52.9549	82.7427	186.06

Table 1: Timings of bucket sort calculations, with the top row being the number of numbers to sort and the leftmost column indicating the amount of buckets used. The red cells indicate ones that took over five minutes, so what got put into that cell is the closest time in that column under five minutes.

Results - Parallel

Runtimes

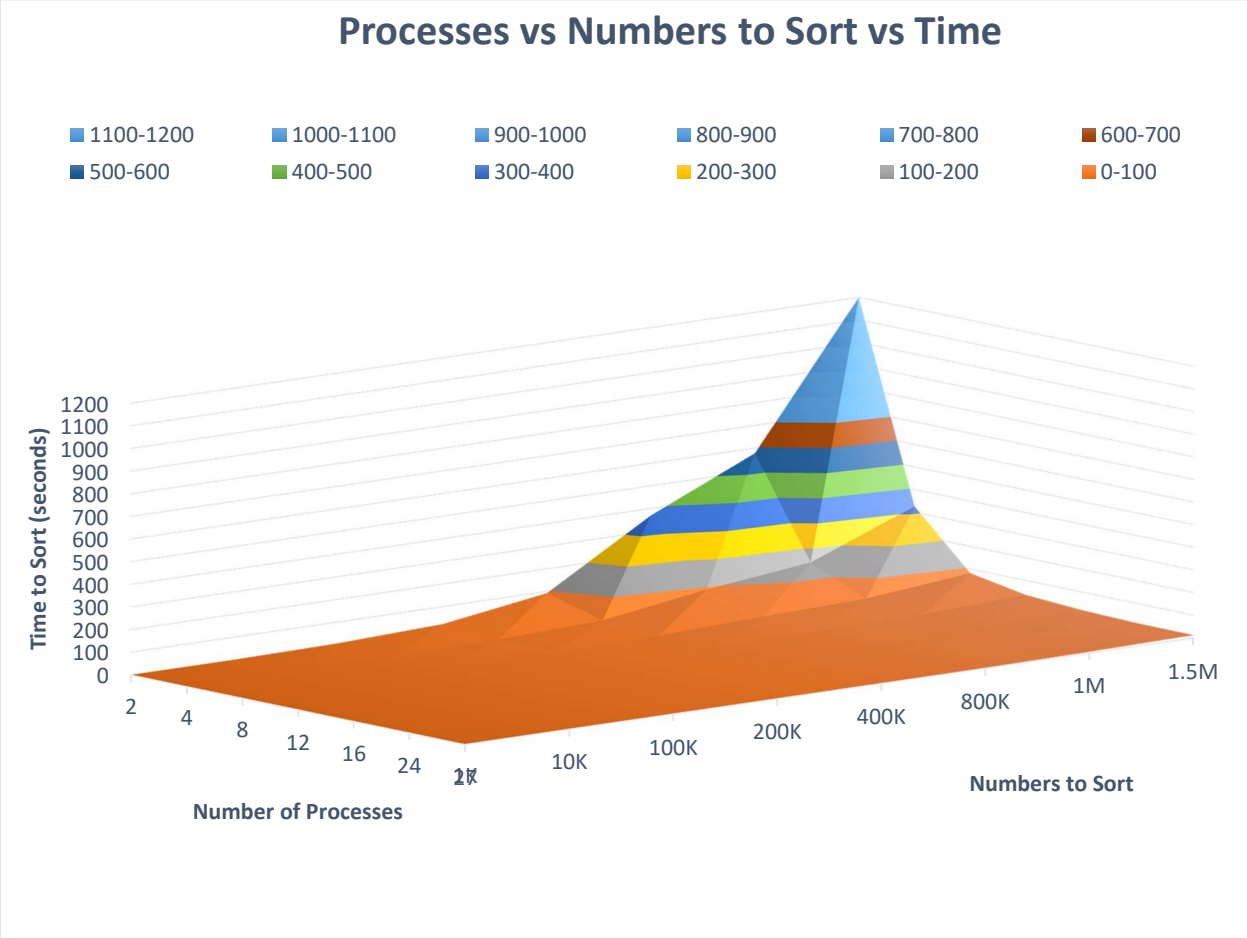

Fig. 2 Graph of parallel runtimes with different amounts of numbers to sort and numbers of processes used in the parallel bucket sort.

	1K	10K	100K	200K	400K	800K	1M	1.5M
2	0.002118	0.067777	5.70005	22.8142	92.3015	369.057	576.943	1200
4	0.000613	0.019866	1.44498	5.77406	22.9258	92.4399	144.716	325.337
8	0.003498	0.010789	0.367435	1.4492	5.79171	22.9341	35.9317	81.5103
12	0.009812	0.023034	0.176181	0.667094	2.59741	10.3053	16.0837	35.8944
16	0.032962	0.081733	0.107824	0.385646	1.46994	5.79746	9.06848	20.2014
24	0.094754	0.02392	0.066362	0.239487	0.708343	5.50641	7.50383	12.9745
27	0.028211	0.029241	0.068665	0.192203	0.681921	2.58447	5.2544	11.7445

Table 2: Timings of bucket sort calculations, with the top row being the number of numbers to sort and the leftmost column indicating the amount of processes used. The yellow cell indicates that the time for sorting 1.5 million numbers using two processes went over 20 minutes, so it was just recorded as 20 minutes.

Speedup

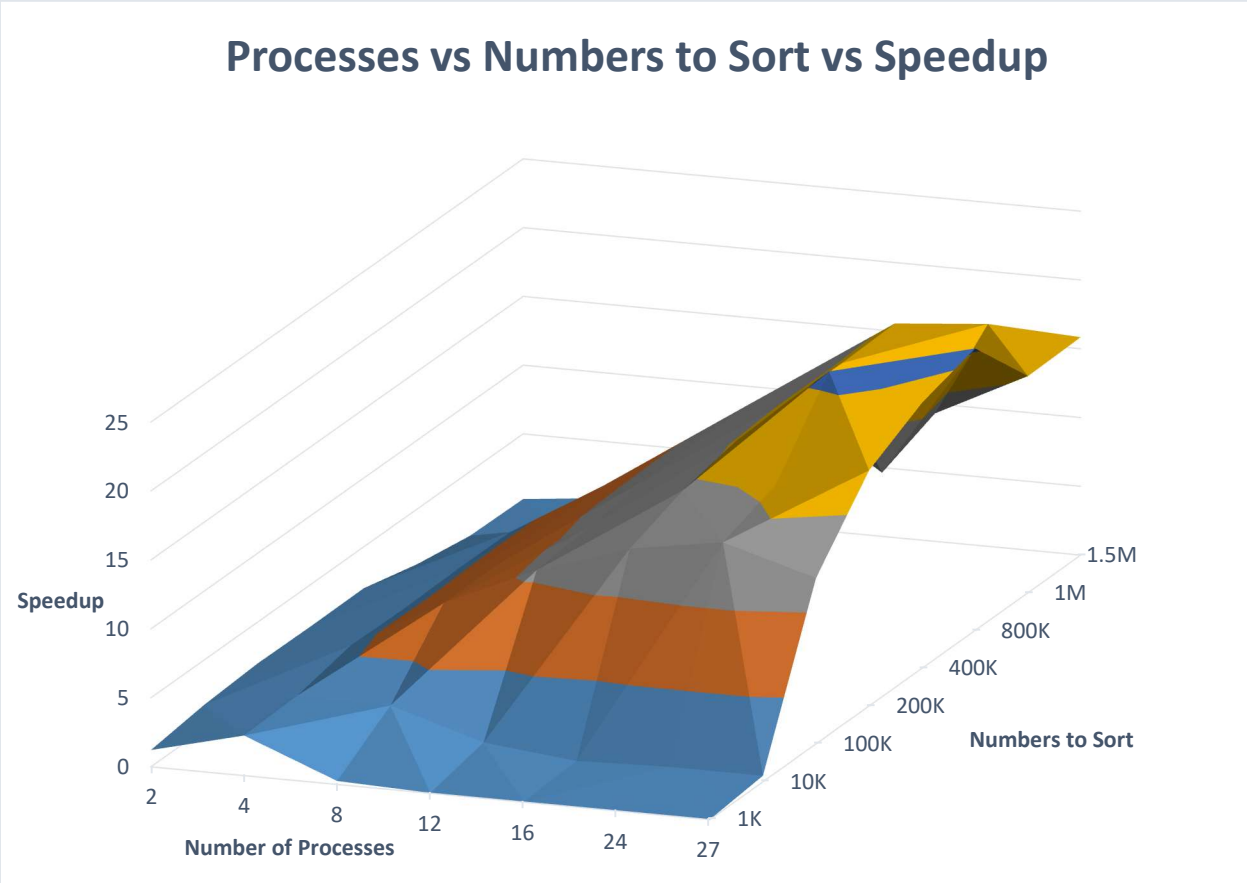

Fig. 3 Graph of speedup with different amounts of numbers to sort and numbers of processes used in the bucket sort.

	1K	10K	100K	200K	400K	800K	1M	1.5M
2	1.21297	1.74506	1.99887	1.95711	1.99112	0.99435	0.32853	0.26201
4	2.89555	3.05059	3.93474	3.86893	3.97083	3.96984	1.30972	0.96661
8	0.24559	2.97015	7.65571	7.69495	7.86434	7.95795	5.27517	3.85831
12	0.02277	0.92181	10.6908	11.0763	11.6756	11.7673	11.7841	8.76179
16	0.00383	0.22894	12.8835	14.3829	15.2206	15.4276	15.4157	15.5688
24	0.00166	0.43357	13.9806	15.4427	20.9173	10.8194	12.4001	16.1688
27	0.00926	0.43404	12.0261	17.10436	19.28449	20.48966	15.74732	15.84231

Table 3: Calculations of speedup of the parallel algorithm,, with the top row being the number of numbers to sort and the leftmost column indicating the amount of processes used.

Efficiency

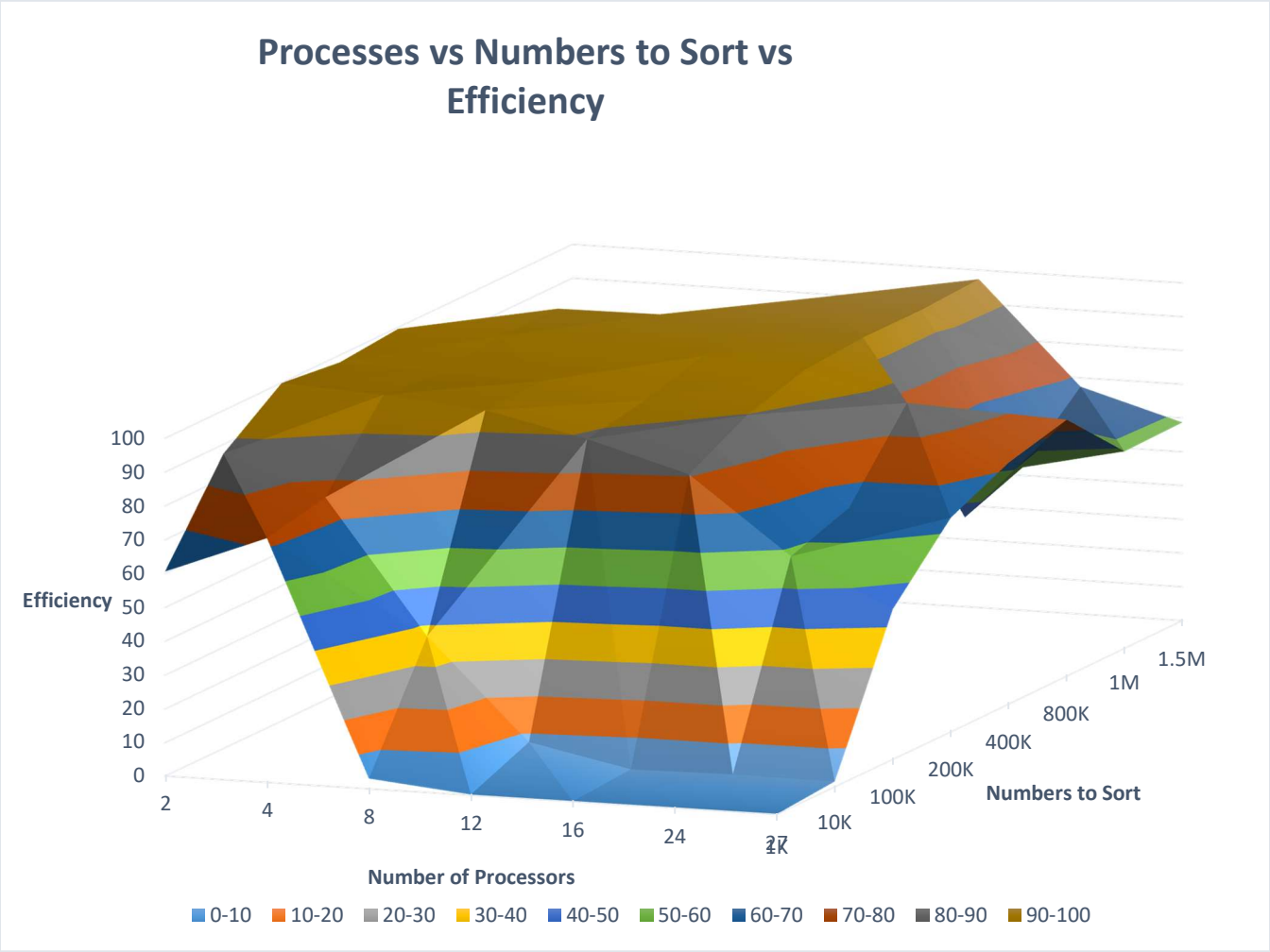

Fig. 4 Graph of efficiency of the parallel bucket sort with different amounts of numbers to sort and numbers of processes used in the bucket sort.

	1K	10K	100K	200K	400K	800K	1M	1.5M
2	60.6464	87.2539	99.9405	97.8556	99.5596	49.7172	16.4267	13.1044
4	72.3899	76.2642	98.3695	96.7243	99.2708	99.2461	32.7441	24.1672
8	3.06961	37.1263	95.6962	96.1869	98.3043	99.4749	65.9394	48.2294
12	0.18934	7.68176	89.0876	92.3066	97.2948	98.0664	98.2071	73.0141
16	0.02381	1.43069	80.5215	89.8912	95.1293	96.4221	96.3485	97.3005
24	0.00686	1.80653	58.2518	64.3429	87.1573	45.0823	51.6705	67.3674
27	0.034134	1.607459	44.54153	63.34949	71.42404	75.8871	58.3239	58.6752

Table 5: Calculations of efficiency of the parallel bucket sort, with the top row being the number of numbers to sort and the leftmost column indicating the amount of processes used.

Analysis

Sequential

As for the sequential algorithm, the main goal was to find out how the runtimes would look with different numbers of items to sort, up to five minutes. There was also the concern about whether the number of buckets used would affect the runtime. With my implementation of bucket sort, the sequential algorithm could only handle 1.5 million numbers until it went over 3 minutes. Even then, anything under 16 buckets would go over 5 minutes. Along those lines, anything under 12 buckets for 1 million numbers and anything under 4 minutes for 800 thousand would go over 5 minutes. In order to still have data points for those cells, the closest time in that same column under 5 minutes was put into any cell that couldn't be calculated since it went over 5 minutes. Those particular cells are highlighted in red on the table provided for the sequential results.

Looking at the data, it is clear that the number of buckets used in the sequential algorithm does affect the overall runtime. Mostly for anything over 200,000 numbers, the more buckets used seems to decrease the overall runtime.

Parallel

Looking at the runtime graph of the parallel implementation, it is evident that as numbers to sort increases the runtime also increases. Notably when the numbers go over 100 thousand, the runtime graph increases by more and more. Although, it is also evident that as the number of processes used increases, particularly once it goes over 12 processes, the runtime decreases drastically.

Speedup was calculated by dividing the sequential time for that number of items to sort (that also used the same amount of buckets as processes used in the parallel version) by the parallel runtime. As the number of processes used goes up, so does the speedup, although it seems to be decreasing as the number of items exceeds a million. There are no cases of superlinear speedup, where the speedup would be a number greater than the amount of processes used.

Efficiency was calculated by dividing the speedup by the number of processes used. An interesting note about the efficiency is that it looks like the only times the efficiency isn't near 100 is when more than 8 processes are used for numbers of items less than 100 thousand. In those cases efficiency drops to nearly zero.