Using statistical analysis to solve the pressing issue of sorting big data

Soumya Mishra

November 2018

1 Abstract

Sorting algorithms are widely used in computer programming but have two key features that present challenges: highly iterative processes requiring intense data churning and large memory requirements to execute these processes. Furthermore, the advent of Big Data has created a greater urgency for a better sorting algorithm with a novel method of sorting these data sets. This was achieved in this research by using the distribution of the data set to presort the data elements into equal-sized buckets, and then individually sort the buckets to produce the final sorted list. This paper presents the unique method - which is called StatSort - through which this is done and the efficiency of this algorithm. Our data has shown that StatSort can be up to 10 times more efficient than merge sort and quick sort and 40 times more efficient than bucket sort. Furthermore, we also found that this unique algorithm has a worst case linear time complexity (O(n)) as well as a space complexity of O(n). With a linear time complexity and ability to use statistical means to sort data, StatSort is the best suited for dealing with big data. In future studies, the effect of implementing this sorting algorithm in current big data processing tools like Apache Spark and Hadoop MapReduce

2 Introduction

Sorting is an essential and fundamental area of study in computer science. Various algorithms have been developed to perform this task. All of these algorithms try to achieve the golden standard: least time and space complexity while still being able to sort the data set accurately. The most used algorithms today are merge sort as well as its variations, quicksort and its variations, and bucket sort. The methods used to sort data sets given these three algorithms were invented over 60 years ago, in the 1940s to 1960s, when data sets were most likely no more than a couple million entries. However, in today's world where Big Data is becoming more and more prominent in any field imaginable, these sorting algorithms, and any other algorithm that relies on these, are becoming dangerously inefficient. While no new sorting algorithms have been created, there have been some slight modifications to the existing ones, most of them just being parallel-processing or using more computational resources. Some of these improvements have made sorting slightly more bearable; however, the need for a new sorting algorithm that is suited for today's data and analysis methods has never been greater. By joining the two fields of distribution analysis and sorting, a new, more efficient sorting algorithm has finally been created: StatSort. This unique sorting algorithm uses the distribution of the data set to pre-sort the data into equal-sized buckets and then sort each of these buckets to produce the sorted list. This paper describes the method by which StatSort sorts data sets and why it is more efficient than leading sorting algorithms like merge sort and quicksort.

3 Background Information

Merge sort was invented in 1945 by John von Neumann. This is a divide and conquer algorithm where a given data set (in the form of an array) is divided into halves until sets of 1 element each are created. Then these sets are sorted back into a complete sorted set of the data. This algorithm has a time complexity of O(n log n) and a space complexity of O(n), making it the most widely used algorithm today. Merge sort requires a second array to help create the many sub arrays, and therefore, has a relatively great space complexity as compared to non-comparison-based sorting algorithms. This is one reason it cannot sort more than a few million numbers.

Another popular algorithm is quicksort, which was invented in 1960. It is also a divide and conquer algorithm and uses recursive partitioning to sort a given data set. The worst-case time complexity for quicksort is O(n^2) however it is practically faster because the inner loop of this algorithm can be implemented on many architectures using most actual data. One way quicksort can be made more efficient is changing the place from the partitioning occurs during each recursion.

There is another algorithm, bucket sort, that only works for certain types of data, but could theoretically be efficient. This algorithm distributes the data elements into 'buckets' that have been assigned ranges of numbers to contain. Then these buckets are either recursively sorted using the same method, or some other sorting algorithm, like merge sort, is used to sort these buckets. Unfortunately, this works ideally for only uniformly distributed data sets and has a worst-case time complexity of O(n^2). If a normally distributed data set were to be sorted using bucket sort, there would be a lot of the data sorted into only a few buckets, leaving all the other buckets virtually empty, wasting space (Bozidar & Dobravec, 2015). This all would make sorting these buckets, and just this data set in general tedious. Surprisingly this is a widely used non-comparison-based sorting algorithm. While these are the commonly used sorting algorithms, these are many more that have similar average and worst cases. Many of these algorithms have best cases of O(n), which is ideal, however, their average cases usually converge to O(n log n). Scientists have been finding ways to have these averages cases converge to O(n) as that is the optimal average case for any system.

To improve these sorts' performances, many variations of them have been created. Corwin and Logar (2012) used linked-lists to make bucket sort more efficient. Many other variations of bucket sort, including that of Hirschberg (2013) include using different GPUs or multiple processing units to sort various buckets at the same time, converting bucket sort into a parallelized algorithm. The parallelized algorithm allows the time complexity to decrease significantly, but also causes the space complexity to increase.

Similarly, there have been methods to improve merge sort and most of them involve around parallelizing the sorting aspect of the algorithm, too, as well as finding new ways to split the arrays. Chong et al. (2012) figured out that merge sort works faster when it sorts greater chunks of data fewer times. By constantly expanding the array sizes in each iteration, the different arrays can be concatenated at various times during the algorithm and this can allow multiple arrays to be joined at once, making the run time more efficient. There will be two moving memory windows – one in registers, which contains the numbers to be sorted, and one in shared memory, which is the sorted array. A complicated algorithm is then used to overlap the windows and transfer number from the register window into the shared one.

All these modifications to parallelize both sorts to decrease the time and space complexity were theoretically proven to work. However, many practical problems arose, making these solutions unfeasible. When using multiple GPUs, there were memory-fetch conflicts because more than one processor was simultaneously accessing the same memory location (Hirschberg, 2013). Linked lists also proved to complicate this problem and even increase the run time by 20% due to memory allocation calls. This wasted unnecessary space and limited the size of the list inputted (Corwin & Logar, 2012). Lastly, when using moving windows to continually expand the array sizes after each pass created a lot of virtual memory problems because the blocks were updated based on two moving windows. One block could not update its register window until everything was sorted and vice versa. This caused a load-balance issue in that if the overlap in the window range was small, some blocks would be used very little or not at all (Davidson et al., 2012).

Due to these issues, these sorting algorithms have stayed the same since they were created, which was more than half a century ago for most of them, and they have not been sorting data any more efficiently. Furthermore, they are only suited to sort small to medium-sized data sets. However, in this current age of collecting data, no data sets are small or medium-sized; most of the data being collected today is considered big data, any data set that overwhelms current IT systems because of their voluminous size. Various big data processing frameworks have been created, like Hadoop MapReduce and Apache Spark, to relieve current IT systems from the immense pressure stemming from dealing with these data sets. These frameworks use a distributed computing environment to process big data more than efficiently than using a normal one-node computing environment. Most of these take sequential, equal-sized chunks of the data and send them out to various nodes (just the first 64 GB, then the second 64 GB, so on so forth), regardless of whether or not they are sorted. However, this very distributed computing environment does not permit sorting because it is based on distributing equal-sized chunks only and then processing those individually. Whenever these voluminous data sets need to be sorted, which is a very common step in processing any type of data, they are sent out to external tools to be sorted. Of course, these use variations of merge sort or quicksort adapted to sort big data. But regardless, this takes up a lot of time and also requires even more processing power because of the methods these algorithms implement. And, until this data is sorted, all processes related to this data set are halted because they require the data to be sorted. Thus, it is imperative that a new sorting algorithm based on novel techniques to sort the data be created.

4 Methods

Method to distribute data elements evenly among ten buckets

The main idea of this algorithm was to distribute the elements evenly between the buckets. For this to happen under any scenario, the ranges of the buckets must be variable and dependent on certain statistical measures of the data set. The simplest and most common distribution – the normal distribution – was the basis of this algorithm as a proof of concept. Initially, we took the probability distribution function of the normal distribution and then integrated it so that each bucket would include 10% of the data; thus, the number of buckets would stay constant while the number of elements in each bucket was dependent on the data set. The possibility of having a fixed number of elements in each bucket and having a variable number of buckets was also investigated, however, it was determined that merge sort was efficient in sorting a million data elements in each bucket. Therefore, we decided to keep the number of buckets constant. This also allowed ease of coming up with a function to determine which element belonged to which bucket. The normal distribution is the simplest out of all in that it's general shape does not change much; it will always retain its bell curve shape and only become wider and narrower, or taller and shorter. Any normally distributed data set can be mapped onto the standard normal distribution because of a statistical measure called the z-score. Due to the ease of normalizing varying statistical measures (mean and standard deviation) in the data sets that follow a normal distribution, this distribution was used as the proof of concept. The algorithm was initially written – in java – to take the integral of the normalized normal distribution from the minimum value of the data set to the number that was being presorted into the buckets. The value would be rounded up to the nearest integer and then the element would be placed into that bucket.

$$\frac{\int_{min}^{x} \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} dx}{0.1n} + 1$$

While this did create rather equal-sized buckets, it was very computationally expensive and it was very inefficient due to the calculation of the integral. Therefore, the bucket numbers (1 to 10) and the upper bounds of the ranges of each bucket were graphed to hopefully produce a function that would output a bucket number quickly while still maintaining accuracy. This function takes in the z-score of the element as its input and accurately outputs the bucket in which the element belongs. This gave a cubic function - this function looked like the antiderivative or area accumulation under the normal distribution curve, potentially giving some insight as to why this formula worked - that was implemented in the algorithm to evenly distribute the elements to their buckets. he output of this function was then rounded down to the nearest bucket number. This formula roughly gave equal sized buckets, and thus this is how StatSort presorts the elements.

$$bucketnumber = 0.3374(zscore)^3 - 0.0456(zscore)^2 + 3.474zscore + 5.3$$

Framework of StatSort

The algorithm passed through the data set three times. The first pass gave the minimum value, maximum value, and sum for the mean. The second pass gave the standard deviation. And the third pass was used to determine the z-score for each number, which would then be used to give the bucket number the data element belongs to. In this third pass, the formula found above was used to determine which bucket the element belonged in. To ensure that these buckets were not drastically different in size, after sorting each element to its respective bucket, the number of elements in each bucket was obtained and stored in a 1D array. These buckets were in the form of a 2D array where the bucket numbers were the columns. After all the elements had been pre-sorted into their respective buckets and it was ensured that the buckets have roughly the same number of elements - plus or minus 0.001 of the total number of elements in the data set - a new method was used to sort the 2D matrix by column. After this was done, the columns were concatenated to produce the final output.

Creating data sets for testing StatSort

Personalized data sets, made of doubles, were made using the Gaussian function in Java where given the desired mean and standard deviation, Java would create a roughly normally distributed data set. Then, we made sure that these data sets were in no way sorted; if the data set had more than 5 consecutive number of 10 sorted numbers, those numbers were modified such that there was no sorted data in the given data set. This was to ensure that the data we collected did measure the worst case time complexity. Furthermore, we wanted to test the effect of the skew of the data sets on StatSort. So, we took these data sets and visually saw their distributions. We then manipulated these data sets such that some of these data sets were slightly skewed to the right and left.

Accounting for skew of data set in StatSort

To account for the skew of the data set in StatSort, several measures were taken. First, given the maximum value from the first pass, we assumed that this would have a z-score of around -3. Thus, using this assumed z-score and the standard deviation of the data set, we figured out the theoretical mean of the data set. This would be "peak" of the data set had it been normally distributed. We could use the standard deviation because this standard deviation does not give any valuable information about the symmetry of skewed distributions. When we obtained the actual mean, we would compare this mean to the theoretical "peak" mean we found. If the actual mean was only approximately 3/4 of the standard deviation from the peak mean, then StatSort was used as usual with no modification. However, if the actual mean was more than 3/4 of a standard deviation away from the mean, then this implied that it was skewed. If the actual mean was less than the peak mean, then it would be considered left-skewed and if the actual mean was greater than the peak mean, then it would be considered right-skewed. If the data was deemed right-skewed, then we added another pass through the data set that would apply a cube-root transformation to the data set and proceed with StatSort starting with the first pass. If the data was left-skewed, then we would apply a cube transformation to the data set and again, proceed with StatSort using this new data set. This was showed to work in terms of sorting; however, the data sets created were not skewed enough to implement this part of the algorithm. There is currently research being done on this aspect of the algorithm.

Obtaining code for other algorithms

To get the code for the other sorting algorithms, we obtained multiple versions of these sorting algorithms from reliable online sources like GeeksforGeeks and TutorialsPoint. We had each of these algorithms sort the data sets and we chose the fastest and accurate version of each of these algorithms to proceed with the research. Most of these codes were from GeeksforGeeks as they had the fastest and more accurate results during preliminary testing.

5 Results

Run time analysis and linearity of StatSort

We first collected the time it took StatSort to sort various data sets, each with a different number of data elements and slightly varying distributions - some were more skewed than others. We conducted a total of ten trials for each data set and averaged them to get the mean run times of StatSort for these data sets. We also calculated the standard errors for each of these trials to see how much the distribution of the data set affected how efficiently StatSort ran. As seen in Table 1, StatSort can sort 50,000,000 elements in roughly 2 seconds. Furthermore, the low standard errors imply that the performance of StatSort does not heavily depend on the distribution; as long as the distribution is somewhat normal regardless of the skew, StatSort will perform consistently. We also used data sets with heavy skew and because we had a second method to use if that were the case, StatSort could sort those data sets efficiently as well. The regression of the mean run times for each

Table 1: Data size and run time analysis for StatSort

Trial	1	2	3	4	5	6	7	8	9	10	Mean	St. Error
1000	0.023	0.023	0.021	0.025	0.023	0.021	0.023	0.029	0.026	0.023	0.023	0.0007
10000	0.031	0.035	0.029	0.031	0.031	0.028	0.031	0.033	0.034	0.031	0.031	0.0006
50000	0.048	0.05	0.047	0.046	0.048	0.048	0.049	0.049	0.05	0.051	0.048	0.0004
100000	0.054	0.055	0.053	0.06	0.049	0.049	0.05	0.053	0.054	0.056	0.053	0.001
500000	0.1	0.098	0.123	0.0975	0.145	0.0978	0.14	0.1	0.1	0.1	0.11	0.0056
1000000	0.192	0.2	0.197	0.189	0.1798	0.185	0.19	0.198	0.192	0.19	0.191	0.0018
5000000	0.265	0.27	0.278	0.265	0.245	0.27	0.265	0.245	0.257	0.265	0.263	0.0031
10000000	0.496	0.512	0.498	0.49	0.487	0.5	0.512	0.541	0.496	0.5	0.503	0.0047
50000000	1.799	1.8	1.798	1.812	1.789	1.83	1.789	1.823	1.799	1.8	1.804	0.0041

of these data sets seemed to increase in a linear fashion, which was further suggested by the linear fit, through Excel, and the R^2 value of 0.9956, as seen in Figure 1. To ensure that these run times did fit only a linear regression, we also fit an n ln n regression through Excel on the data and obtained an R^2 value of 0.485, as seen in Figure 2. This disparity between the R^2 of the two regressions was enough to suggest that a linear time complexity for StatSort was possible. However, we wanted to ensure that this was statistically significant. We conducted a chi-squared regression goodness-of-fit test where we compared the experimental time complexity and the theoretical time complexity of O(n). As seen in Figure 3, the two regressions seem to align with each other, which was further suggested by the high p-value of this test. Because our null hypothesis was that the experimental and theoretical time complexity regressions would be equal and our high p-value, we could not reject this null hypothesis, gathering up more evidence that the time complexity of StatSort was indeed linear.

Comparison of StatSort to other major types of methods and sorting algorithms

Now that this was established, we wanted to compare StatSort's performance to that of other sorting algorithm that had unique methods to sort the data. We used the same data sets that we used in the initial data collection process for StatSort and recorded the times it took for these algorithms to sort the data sets. Again, ten trials were repeated for each algorithm on each data set and these trials were then averaged, giving us the values in Table 2. As seen by the last row, StatSort seemed to have sorted the largest data set the fastest, followed by bucket sort, then merge sort, then bead sort, then radix sort. Some algorithms, like bubble sort and quick sort failed to sort the 50,000,000 element data set for these algorithms ran into a heap error. Some algorithms did not encounter these heap errors, but took on the order of a couple of minutes to sort the data set. This suggests that StatSort might be one of the more faster sorting algorithms due to its distribution

Figure 1: Experimental time complexity given run times for various sized data sets

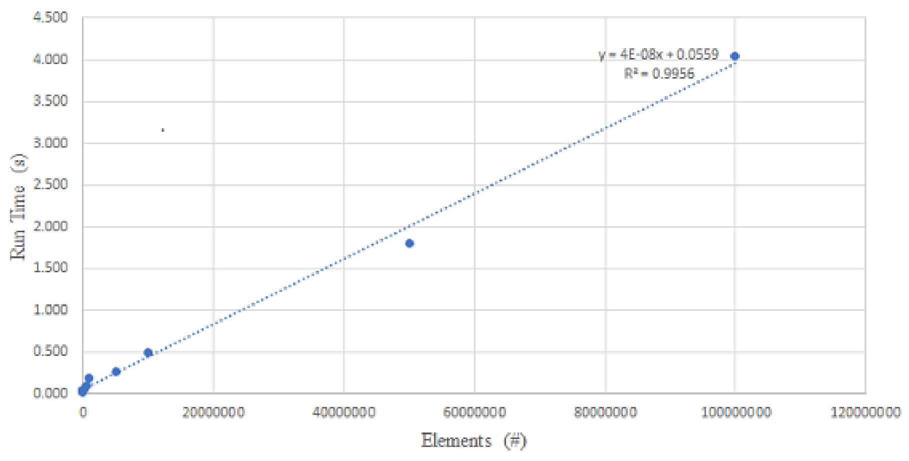


Figure 2: Experimetnal time complexity with n log n fit

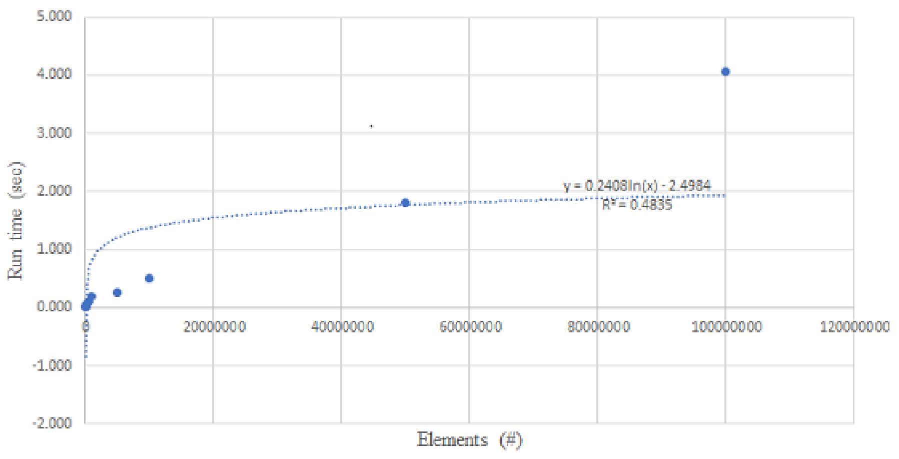
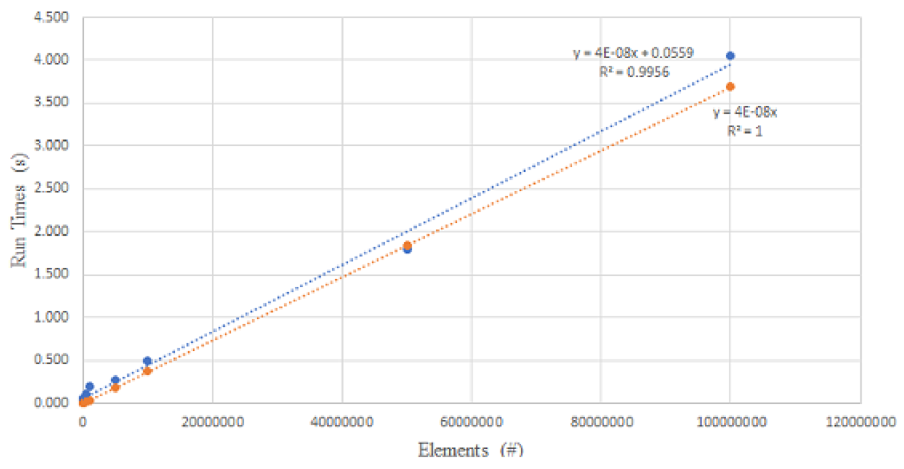


Figure 3: Comparison of theoretical and experimental time complexity



method. Bucket sort also has a distribution method; however, due to the uniformity of the distribution of the elements into the buckets, it is not a heavily relied upon sorting algorithm. And given that this sorting algorithm has a linear time complexity given a uniformly distributed data set, it is not a surprise that because we were able to distribute the normally distributed data sets evenly between the buckets we could reach a linear time complexity. We were also interested to see how much more efficient StatSort was compared to the three fastest algorithms used in this experiment. As seen in Table 3, as the data set size increases, the percent efficiency of StatSort compared to merge, bucket, and radix sort also increased on the whole. StatSort reached up to percent efficiency of approximately 380% when compared to merge sort, which is the most widely used sorting algorithm today, and approximately 1000% more efficient than radix sort, which is mostly used to lexicographical data. Thus, due to this linear worst case time complexity StatSort was seen to perform much better - three to ten times better - than the current leading sorting algorithms. And, to clear up any doubt about the linear time complexity of StatSort, we conducted the chi-squared regression goodness-of-fit test on all the sorting algorithms and compared their experimental time complexities given from the data to their stated worst case time complexities. Again, we obtained p-values too large to reject the null hypothesis that the experimental and theoretical time complexities of these algorithms with the collected data were equal. Thus, this strengthened our belief that StatSort did have a worst case linear time complexity.

Comparison of sorting methods

This exemplary performance of StatSort can be attributed to the limited number of passes it has to make through the data set - a maximum of four passes - as compared to the iterative passes through the data sets for the other comparison-based algorithms. Merge sort, which implements a divide and conquer approach to sorting, passes through the data log(n) times because it needs to repetitively split the data into the smallest unit for it to sort the data set. The other comparison based sorts - heap, insertion, bubble, pancake, and quick sort - need to make at least n passes through the data set because they require comparing each element to every other in the data set when trying to figure out where the element belongs. The

Table 2: Data size and run times for all sorting algorithms and methods

	Stat	Merge	Bucket	Radix	Heap	Insertion	Bubble	Pancake	Quick	Bead
Method	Dist	Div & Conq	Dist	Non-comp	Select	Insert	Exchange	Comp	Partition	Non-comp
1000	0.024	0.034	0.031	0.034	0.048	0.040	0.034	0.036	0.026	0.033
10000	0.031	0.037	0.027	0.051	0.051	0.049	0.235	0.133	0.036	0.026
50000	0.049	0.042	0.040	0.077	0.207	0.374	2.389	1.682	0.176	0.043
100000	0.053	0.056	0.047	0.162	0.229	2.231	7.357	5.224	1.010	0.048
500000	0.110	0.139	0.092	0.292	0.358	4.538	20.398	10.211	2.54	0.170
1000000	0.191	0.462	0.231	1.254	0.695	8.6429	60.132	22.698	10.631	0.442
5000000	0.262	0.762	0.642	0.292	10.059	150.041	130.041	30.646	20.644	0.941
10000000	0.50318	1.718	1.745	6.547	19.44	23.068	389.657	50.507	70.455	1.768
50000000	1.804	8.655	8.425	21.551	66.966	160.232	N/A	80.268	N/A	10.462

Table 3: Percent efficiency of StatSort compared to three fastest sorting algorithms

Data Size	Merge	Bucket	Radix
1000	41.67	29.17	41.67
10000	19.35	-12.90	64.52
50000	-14.29	-18.37	57.14
100000	5.66	-11.32	205.66
500000	26.36	-16.36	165.45
1000000	141.88	20.94	556.54
5000000	190.84	145.04	1132.82
10000000	241.48	246.85	1201.33
50000000	379.77	367.02	1094.62

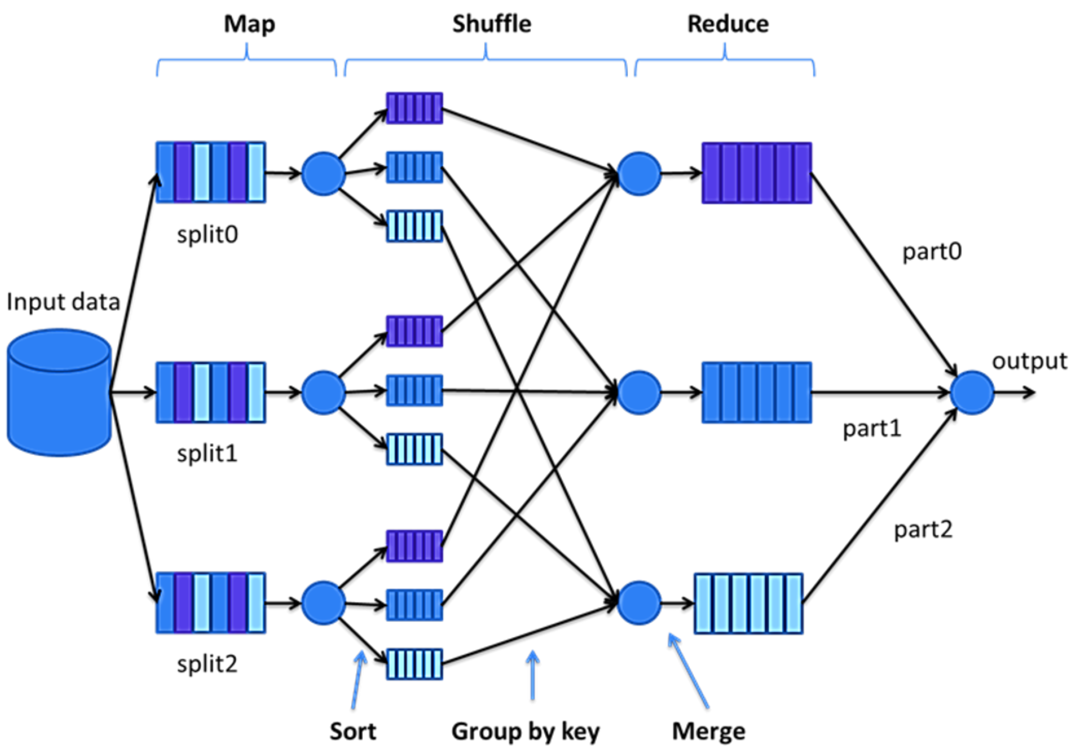
only difference between these methods is simply how they modify the array as they make the sorted data set. Bubble sort exchanges data elements, insertion sort simply shifts the array by an index to put in the element in its respective position. The non-comparison based sorting algorithms, on average, seemed to be more efficient than their comparison based counterparts, as seen in Table 2. Bucket sort is known to be linear when sorting uniformly distributed data because each data element needs to be compared to only a set number of values - which ends up being however many buckets are being used in the algorithm at the time - through each pass of the algorithm to produce the final array. Radix sort uses the place values of the number to produce the final lists - it first sorts it by the ones, tens, hundreds places. Through this, radix sort has minimal passes through the data set and only compares a set number of values at a time. Bead sort uses an abacus method to sort the values. The data elements are essentially aligned rows and through each iteration, the beads are slid down one row until they can slide no more. This does not require any type of comparison, just a measure of where it is in the array. This shows that non-comparison based sorting is theoretically the more efficient methods of sorting; however, these non-comparison based algorithms require certain conditions in ordered to be used to the best of their functionality, making them unappealing. However, this is not the case for StatSort as seen by its consistent performance in sorting data sets of varying distributions.

6 Conclusion

The statistical evidence supports the author's contention that a new sorting algorithm, StatSort, indeed has a worst case linear time complexity which is very compelling and potentially exciting for end users in need of a faster sort method. The comparable data collected for the run times of merge sort, parallel sort, and bucket sort are consistent with their theoretical time complexities as is the author's new StatSort being consistent with a worst case linear time complexity instead of the worst case non-linear time complexity of the others.

Furthermore, a somewhat theoretical proof that StatSort is linear starts with the initial two passes through the data set in one for loop to calculate the mean and standard deviation of the data set. There is a second non-nested for loop dedicated to presorting the elements into their respective buckets; this is the third and final pass through the data set. As of now, the theoretical time complexity is O(n) because there are no nested for loops. Lastly, the parallel computation to sort multiple columns at once also leads to a linear, or even faster, time complexity. These three main steps yield a linear time complexity. Further proofs on this algorithms linearity will be developed; however, the statistical analysis and logical progression of its linearity support the hypothesis that this distributed method of sorting is much more efficient than leading sorting algorithms like merge sort, parallel sort, and bucket sort. This conclusion could be used to implement a solution that could make processing big data much more efficient that it currently is.

The diagram below from Soft Computing and Intelligent Information Systems represents the framework of big data tools like Apache Spark and Hadoop MapReduce. The basic ideology behind these tools is similar to that of merge sort: divide and conquer. These tools implement a distributed computing system too to make processing big data efficient. Essentially, when a data set is fed into these tools, equal chunks of data are sent to jobs (different nodes on clusters). The first X GB of data is sent the first mapper job, the second X GB of data is sent to the second mapper job, so on so forth. These equal sized chunks are just sequential, independent of the actual sorted status of the data. Then, each of these mapper jobs are sorted independently of each other and process their respective chunks of data however the user wants them to be. If the data does not need to be sorted, which is relatively rare, these mapper jobs map their processed data onto the shuffler jobs which rearrange all the data until the desired output is produced. A lot of times, the sorting of the whole data set occurs at the shuffler. The shuffler jobs wait until all the mapper jobs have been received and then these jobs send the data out to an external sorter, which is, in simple terms, a large computationally expensive tool that uses distributed versions of merge sort or bucket sort, or just plain brute force, to sort the whole data set. As discussed above, these distributed versions are not as efficient as developers want them to be. Furthermore, while external sorter is sorting this data set, the tool is simply not doing anything; it's just waiting for the sorting to be completed so that it can produce the output after everything has been sorted. Therefore, a lot of time is wasted from the time when the shuffler has to wait for all the mapper jobs to finish and send their jobs until the time when the shuffler receives the sorted data from the external sorter. This shuffler step is the most computationally expensive step of this framework. However, this framework could be modified using the ideology



behind StatSort, which has proven to work better than merge sort. This ideology will take full advantage of the distributed system to sort data. Essentially, there will be two passes through the data set where the statistical information will be calculated. The mapper jobs will act as the buckets in respect to the algorithm. The algorithm will pre-sort the elements to their mapper jobs. Then, because these jobs are all pre-sorted and equal-sized, the jobs can sort their respective chunks of data independent of the other jobs. Once a mapper job is done sorting the data, it can immediately send it to the shuffler, which can immediately produce an output. The shuffler will not have to wait for all the mapper jobs to be completed before it can start sorting because of the pre-sorting step. This will save valuable time because the whole external sort will not even have to be implemented. Less space can be taken up by the data set in the distributed system while still completing the same amount of work. The rest of the framework can work as is without much modification. This will be most effective implementation of these tools because each mapper job will be independent of the other in terms of sorting, which is the heart of many data analysis tasks whether it be related to machine learning or graphical processing.

We are currently implementing this algorithm to making processing big data much more efficient and to allow current big data frameworks to sort data, which they have not been able to do. No preliminary data is available yet, thought we hope to start collecting some soon.

7 References

Wang, L., et al. (2012). G-Hadoop: MapReduce across distributed data centers for data-intensive computing. Future Generation Computer Systems. doi:10/1016/j.future.2012.09.00

Monaknov, A. (2016). Composable multi-threading for python libraries.

Goel, N., Laxmi, V., Saxena, A. (2015). Handling multithreading approach using java. International Journal of Computer Science Trends and Technology.3.2.

Hai, H., Guang-hui, J., & Xiao-tian, Z. (2013). A fast numerical approach for Whipple shield ballistic limit analysis. Acta Astronautica. 93. 112-120. Retrieved from http://dx.doi.org/10.1016/j.actaastro.2013.06.014

Dehne, F., Zaboli, H. (2016). Parallel sorting for GPUs. Emergent Computation. 293 – 302. doi:10.1007/978-3-319-46376-6_12