Soumya Mishra
06/04/2018
Final Junior Research Paper

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

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 better sorting algorithms. Given the nature of sorting, it is self-evident that sorting 1TB of data would be harder than sorting one thousand ordered sets of 1GB data each.

The objective of this research was to create an algorithm to distribute the data into pre-ordered sets, of the same frequency by using statistical means. The framework of this algorithm consisted of four major steps: collect statistical information of the data (mean, standard deviation, distribution type, minimum, maximum), assign each element in the data set to one of the ordered buckets, sort each bucket, and then concatenate the buckets into a sorted list. The crux of this algorithm is assigning each element to a bucket. Three options were evaluated for the same.

The first option involved using the mean and standard deviation to determine bucket number and was significantly more efficient than current sorting algoirhtm as shown by efficiency rates up to 200% as well as a p-value less than 0.001. Current research to sort other types of distributions is ongoing.

The second option involved integrating the function of the distribution curve - that fit the data set - from the minimum value to the value being assigned to a bucket. The third approach involved splitting up the data set into quantiles and then sorting them as if they were the buckets themselves. Research on the second and third techniques is ongoing.

Soumya Mishra
06/04/2018
Final Junior Research Paper

Introduction:

Any data set that is so voluminous and complex, such that traditional data-processing applications are inefficient and inadequate to analyze or process them in any way, is considered as big data. The three Vs – volume, velocity, and variety – are used to characterize various levels of big data. Volume refers to how much data a user is dealing with. Velocity refers to how fast the data is incoming. And variety is what types of data are being collected. Through the past few decades, the volume, velocity, and variety of data have been increasing only to place severe stress on current IT systems. This has called for the creation of solutions to help store, maintain, and process these data sets using the least amount of resources, time, and cost.

One such solution is the Apache Spark. Apache Spark is an open-source distributed cluster framework with in-memory data processing engines that are optimal for ETL, analytics, and machine learning on types of data in different programming languages: Scala, Python, Java, R, and SQL. This type of framework allows for it to provide massive storage for any type of data, enormous processing power, and the ability to handle limitless concurrent tasks.

It is the most efficient software for processing big data because it automatically distributes the data and tasks between multiple machines and utilizes the underlying parallelism of CPU cores. As part of its process, it can handle failures at the application layer, making it the most reliable software as well. The user does not need to worry about preprocessing the data before storing the data using Spark, which promotes its efficiency. Furthermore, because it is an in-memory computing engine, it can provide better performance for certain applications when compared to other big data frameworks like Apache Hadoop.

Both Hadoop and Spark have main stages: the shuffling – which includes the map and reduce jobs – and sorting stages. These have similar functions in both Hadoop and Spark; however, Spark's implementation is much more efficient due to the heavy emphasis on limiting the use of external memory for each cluster. The basic backbone structure for these implementations include splitting up the data into equal chunks, sending these chunks to mapper jobs, which then send the manipulated data to the shuffler, which then sends the processed data to the reducer jobs to produce a final output. In Spark, the number of mappers and reducers is much higher because Spark, unlike Hadoop, does not merge these files into larger partitioned ones. Unlike Hadoop, Spark requires all the shuffled data to fit into memory of the corresponding reducer task and throws an out-of-memory exception if this cannot happen. This has proved a challenge for developers. Furthermore, Spark's reducer jobs do not push data from the mappers to the reducers whether or not they are complete. They are considered to be pull operations; the reducers will pull the data from mappers when the operation is completed. This makes Spark a much more efficient tool to deal with big data.

The sorting operation then sends data from the shuffler to the reducer jobs and produce the output. Sorting, however, is one of the most challenging because there is no reduction of data along the pipeline.

There are various sorting algorithms that could be implemented to solve the issue of sorting this data. Some of these sorting algorithms have stood out like merge sort and bucket sort for their superior performance. However, there are two key features of every sorting algorithm that increase their time and space complexities - a) highly iterative processes requiring intense data churning and b) large memory to execute these iterative processes. As a result, sorting algorithms take long to execute and put severe stress on system resources.

Each algorithm has a worst-case and best-case scenario which are notated using Big-O notation; this is the theoretical measure of the execution of an algorithm and is also sometimes called the time complexity. The worst-case is the time it takes for the algorithm to sort an unsorted array while the best-case is the time it takes for the algorithm to sort a sorted array. Sorting algorithms are characterized by their worst-case scenarios as the best-case scenarios describes the time taken to sort under optimal conditions, which are almost never present due to variability of data sets.

Each sorting algorithm also has a space complexity, which also uses Big-O notation, and this is the theoretical measure of how much space the algorithm will take while it is sorting. The time and space complexity of the different sorting algorithm are not dependent on the language they are being programmed in because these time complexities are relative to one another, not the various laptops and languages they can be programmed on (Dehne & Zaboli, 2016). The nine sorting algorithms can be separated into two types of sorting algorithms: comparison based and non-comparison-based sorting algorithms.

Comparison-based sorting algorithms compare one element with other elements in the same data set to figure out the most suitable position to place the element to sort the data. The most efficient and respected comparison-based sorting algorithm used today is merge sort. This sort recursively merges multiple sorted subsequences into a single sorted sequence. It has a best-case and worstcase of O (n log n), where *n* is the number of data elements. Merge sort is more appealing than other sorting algorithms because it can handle variable-length keys, which are numbers of different bits, and requires less space than other comparison-based sorting algorithms (Davidson, Tarjan, Garland, & Owens, 2012).

On the other hand, non-comparison-based sorting algorithms use various techniques to approximate the final position of elements in the data set without comparing them. One only needs the value and address of the element. For example, in the bucket sort algorithm, there are buckets, or arrays, with predetermined ranges. When a data set is inputted, it determines within which range a certain number lies and places that number into that bucket. It then uses merge sort to sort the individual buckets, and then concatenates them into one sorted sequence. This has a best-case scenario of O(n+k) and a worst-case scenario of O (n^2), with *n* being the number of elements and *k* being the number of threads, or the number of buckets.

Both merge sort and bucket sort are the best of their types, however, they do come with serious flaws that can affect sorting large data sets. 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. Bucket sort expects the given data set to follow a normal distribution and cannot be manipulated into sorting differently distributed numbers. In a normal distribution, most of the data will lie within 1 standard deviation from the mean. This causes a lot of the data to lie within a few buckets, making the sorting of those buckets tedious and long. Furthermore, a lot of the other buckets are left empty and the space complexity is increased (Bozidar & Dobravec, 2015). Hai, Guang-hui, & Xiao-tian, (2013) have tried to make bucket sort more efficient. They hypothesized that the data points should be more uniformly distributed between the buckets. However, this caused a lot of buckets to be created, thus increasing the space complexity.

To improve these sorts' performance, many variations of these sorts have been created. . Corwin and Logar (2012) used linked-lists, or lists that are connected to their parent or daughter lists or arrays, to make bucket sort more efficient. Many of the variations include using different GPUs,

or graphical 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. Hirschberg (2013) proposed using multiple processing units to parallelize the sorting of the buckets of bucket sort because all the processors have access to a common memory and have their own local memories, allowing them to be synchronized while they sort their own buckets. Another group of researchers decided to divide the data into multiple buckets and use different GPUs to sort the various buckets using radix sort. This idea proved to work faster than the current bucket sort; it was also non-dependent on the distribution of the data set, thus making it practical in more sorting situations (Dehne et al., 2016).

Similarly, there have been methods to improve merge sort as well 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). Furthermore, using radix sort to sort data sets using GPUs was proven to be inefficient because radix sort only converts lexicographical and fixed-length keys, or keys that are all the same bit. 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).

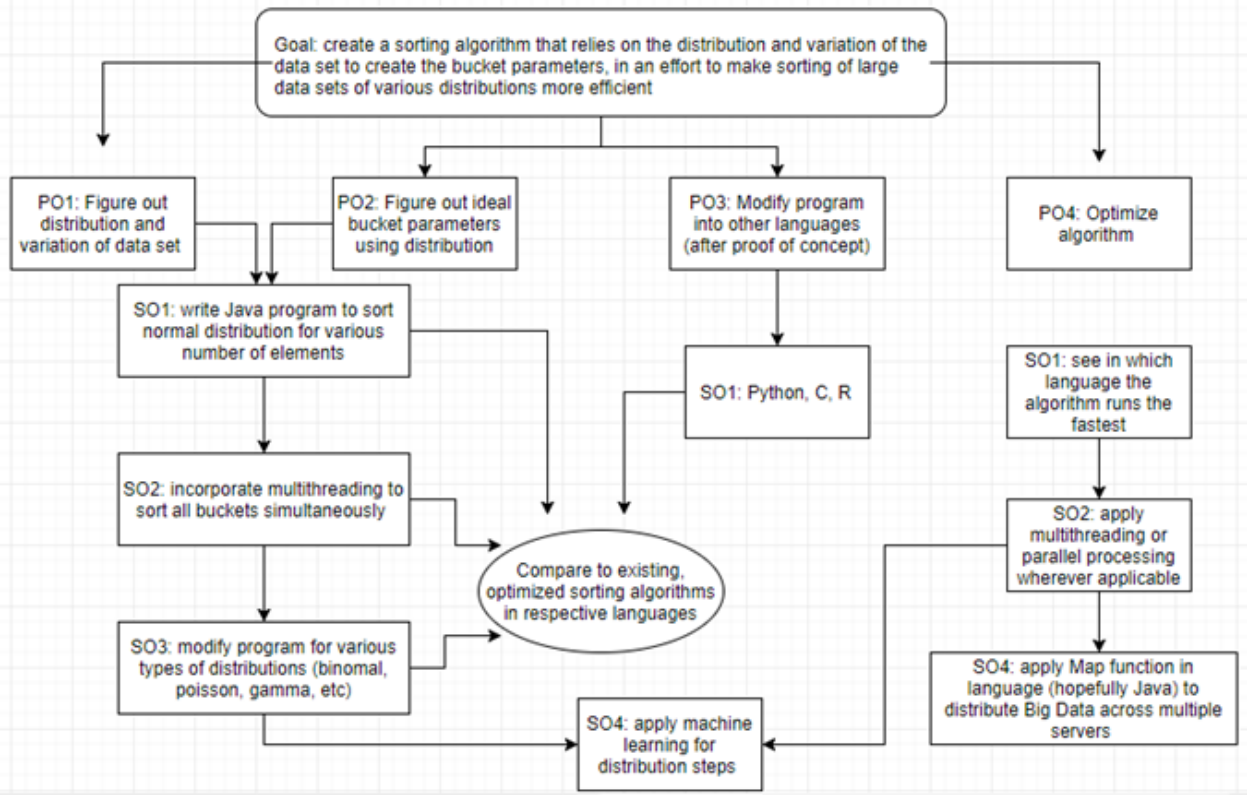
There have been proven methods of how to parallelize the computations of certain algorithms as well as distributing the data set across multiple servers to decrease time and space complexities.

Hong (2012) addresses the memory-allocation problems presented by Hirschberg (2013) during his research as he uses OpenMP API while bucket sort is running. OpenMP API is a shared-memory application programming interface and is built upon previous works on trying to parallelize sorting (Jocksch et al., 2016). Initially, the algorithm will parallelize the bucket sort by splitting n elements into k sets, each having n/k elements. Then, bucket sort will sort the numbers normally, create various sorted threads, and then concatenate those threads. While in this approach there are fewer elements in each sort and thus allows a faster run time, the threads do not all work at the same time. OpenMP API describes how the work is to be split amongst the various threads that will perform bucket sort on different processors. One major benefit of this

solution is that OpenMP can be run on many different platforms. This greatly improves the time complexity, making any algorithm that uses this more efficient.

A key observation is that none of the sorting algorithms or modified algorithms developed to date use the statistical distribution of the data to their advantage for faster performance. This research shall explore the possibility of using statistical distribution type to separate the data into almost evenly distributed buckets, which was suggested in the study Hai et al. (2012) performed. The buckets can be sorted separately using merge sort, and then be concatenated together to provide a final sorted list. Sorting in smaller groups will significantly reduce the usage of system resources as well as result in faster performance. Furthermore, the number of passes needed to be made in this algorithm will be significantly less than those of current sorting algorithms (about an average of 3 passes as opposed to n number of passes where n is the number of elements). Developing a faster sorting algorithm will have a snowball effect of improving other computer algorithms.

Methods:



There are a few major steps involved in programming this algorithm. First, the distribution and variation of the data set will have to be figured out to set the correct parameters for each bucket. This will be done by using values that indicate variation for each specific type of distribution (like standard deviation for normal distribution) to get an idea of the spread of the data. If the distribution of the data set is very irregular, then a pattern or formula will be created to figure out a vague idea of the shape, distribution, and variation of the data set. For example, the mean of the data set and the standard deviation of the data set could be used to figure out the basic shape of the data set, and then the most similarly-shaped regular distribution could be used to figure out a value for the spread of the data set. These will then be used to set the parameters of the buckets in which each bucket will contain the same number elements (pass 1). So, based on the distribution, it will be easier to determine where the data is concentrated most. Thus, it will only be logical to have a greater number of buckets within that distance from the mean.

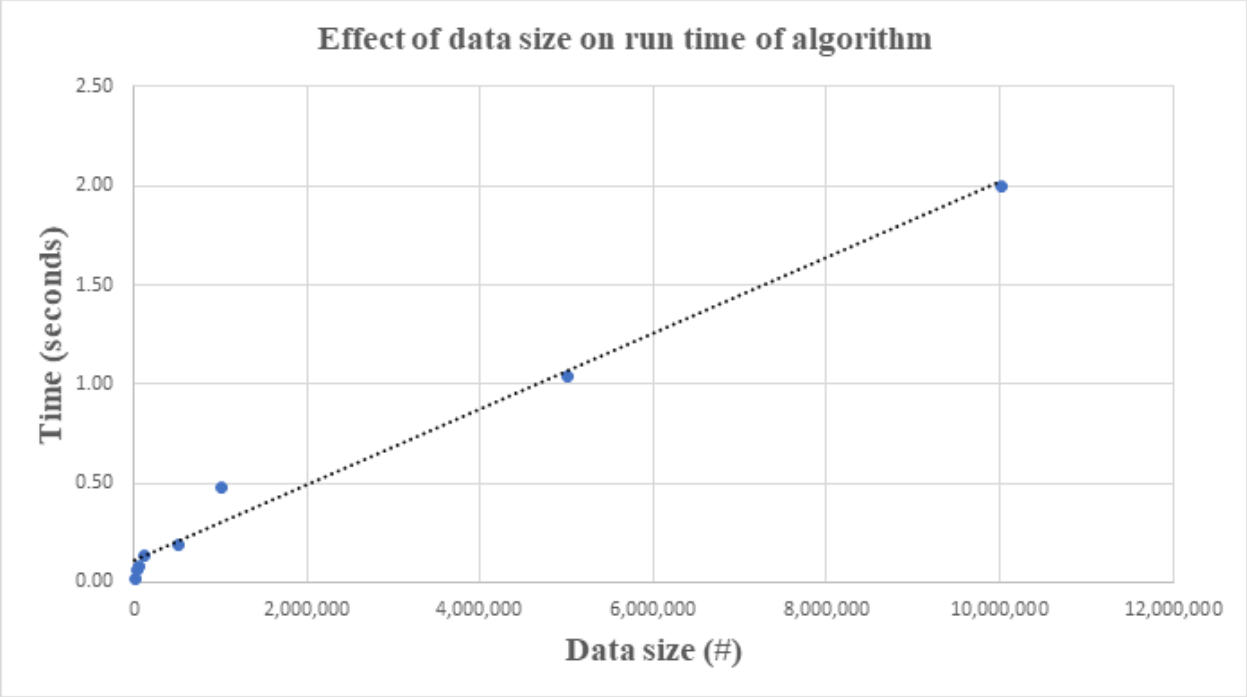
Then, as the data points get further away from the mean, the bucket parameters will get larger. This will ensure that all the buckets created will be used instead of creating extra buckets that only take up space, like in the study Hai et al. (2012) performed. The greatest number in each bucket will be less than the smallest number in the next bucket, thus the buckets themselves will be sorted (pass 2). After these buckets are sorted using merge sort, the arrays can then be concatenated into one sorted array (pass 3). This algorithm will then be compared with a sorting algorithm (Merge sort/Parallel Sort and bucket sort) that I write myself in Java (this allows for consistency and accurate comparisons for both of these algorithms will be unoptimized) and some preliminary data will be collected. I will then collect data by feeding the data sets into my optimized algorithm and then use the mean run time for each set of data sets to determine the Big O Notation for my algorithm. I will plot the points on excel or Mathematica (whichever can allow me to create the most accurate and precise formula) and then figure out the regression that best fits the run time. The data set fed into the algorithm will just be a randomized array created in the beginning of the program. Then, to expand my algorithm to work for different distributions, I can integrate the best-fit function of the distribution curve the data set fits from the minimum value to the value being assigned to a bucket. To take into account the assumption that the area under the curve corresponds to the number of elements that lie within that segment, I will modify the function, and it should return the bucket number of the value when the element is given to it. I can also split the data up into quantiles, which would by nature contain the same number of elements. This is similar to how a box-and-whisker plot uses the median of its quartiles. These quantiles can be treated as their own buckets, which can then be sorted. Because of Java's rudimentary nature, it is plausible that integrating a function or splitting the data up into quantiles may not be viable options. Apache Commons may help; however, the computations

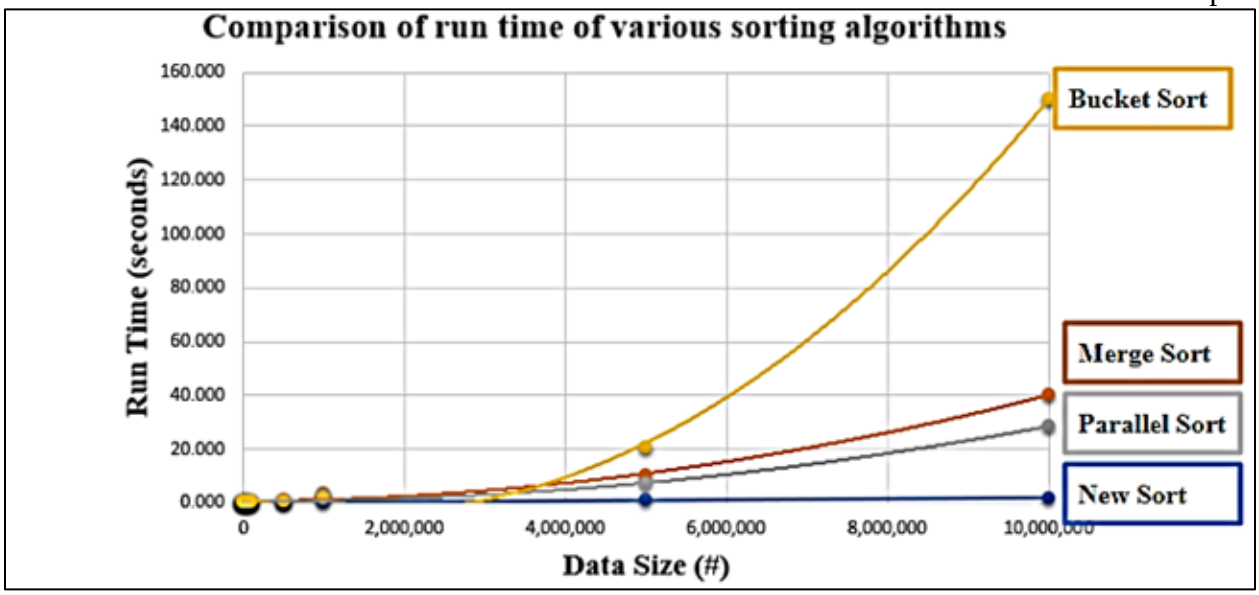
themselves may take time. If this is the case, I will program those algorithms in Mathematica, and Python if Mathematica does not work. Regardless of the language I program the algorithm in, I will collect data in the same manner as in Steps 1 and 2 and compare which language my algorithm works fastest in. I will work to optimize the algorithm in that language and further test the efficiency by having it sort official Big Data. I will learn multithreading in Java as well and compare which program runs my algorithm faster. I will then collect data by feeding the data sets into my optimized algorithm and then use the mean run time for each set of data sets to determine the Big O Notation for my algorithm. To finally prove that my algorithm can increase the efficiency of existing algorithms with a near linear run time, I will embed my algorithm into the framework of Apache Spark and then record how long it takes to sort data sets of various distributions and then see whether it works faster - and if so, by how much - than the original framework of Spark.

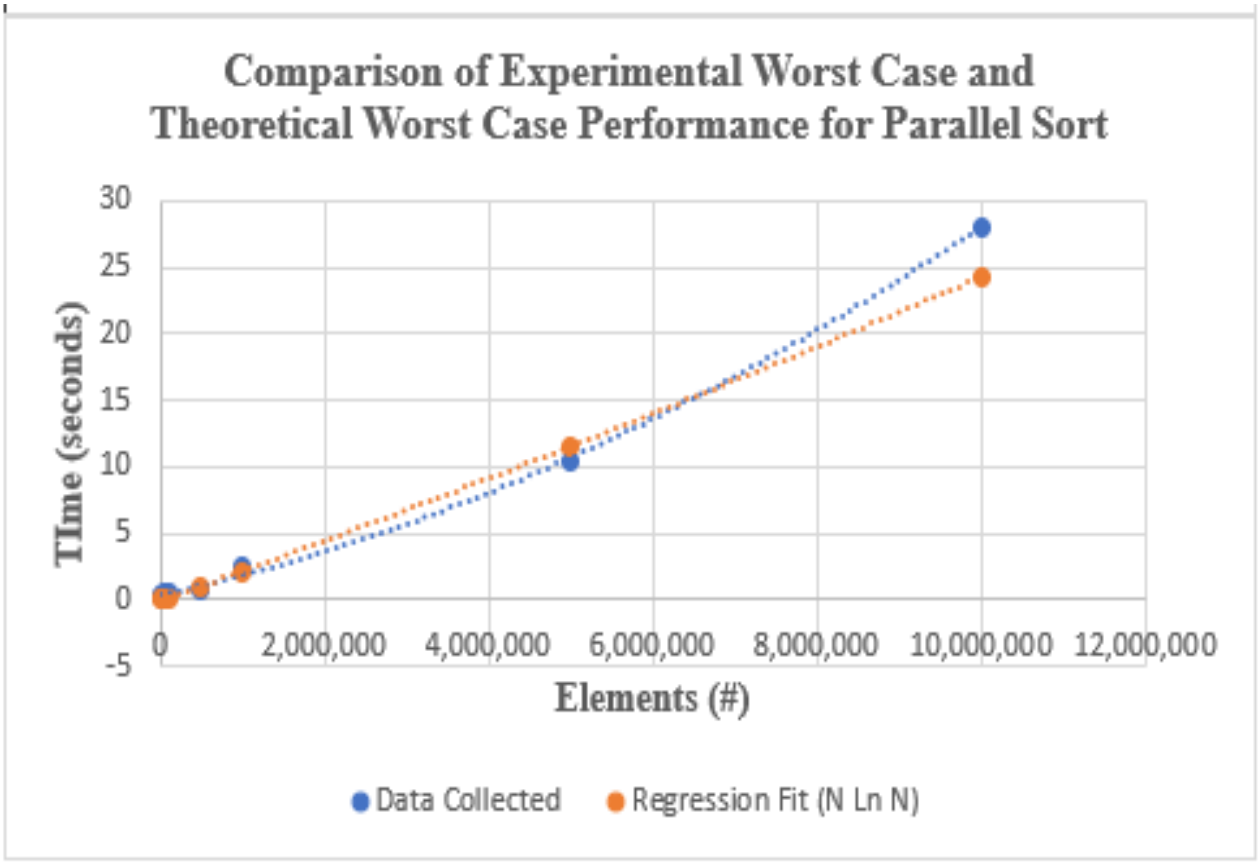
Soumya Mishra
06/04/2018
Final Junior Research Paper

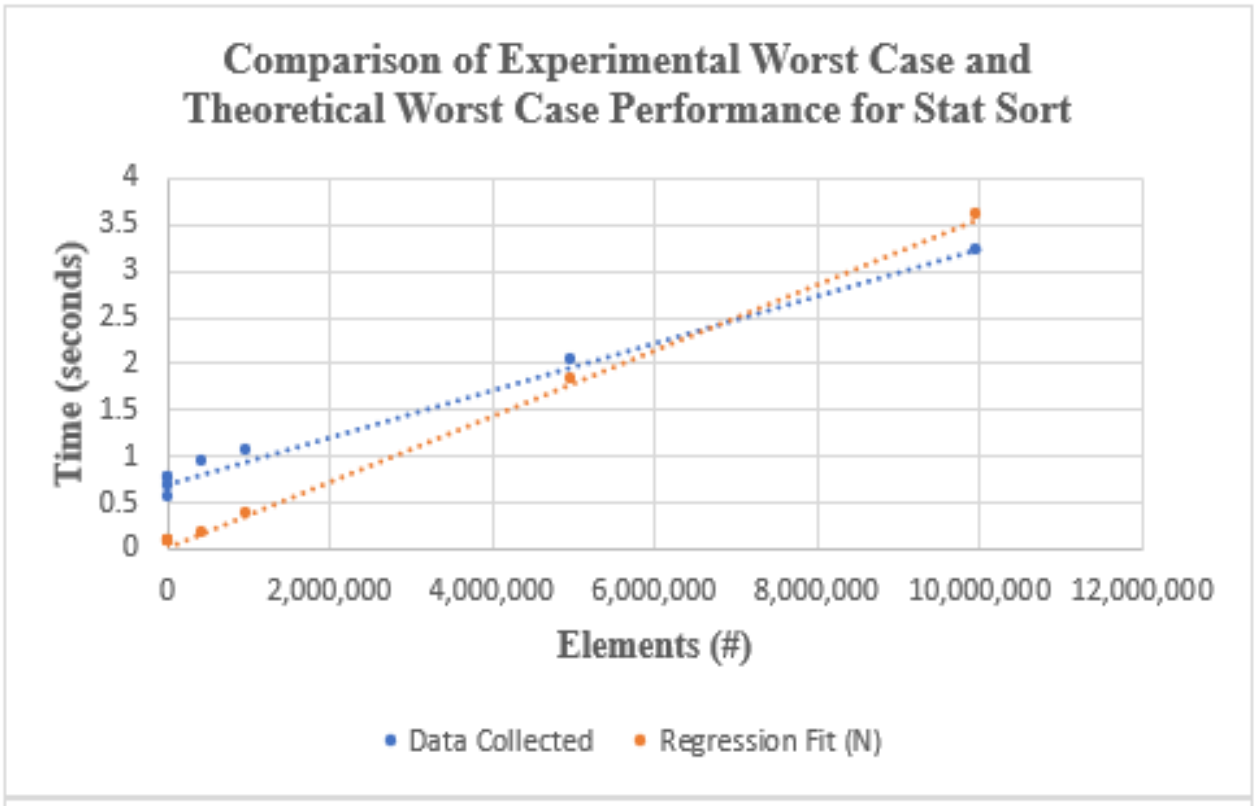
Results:

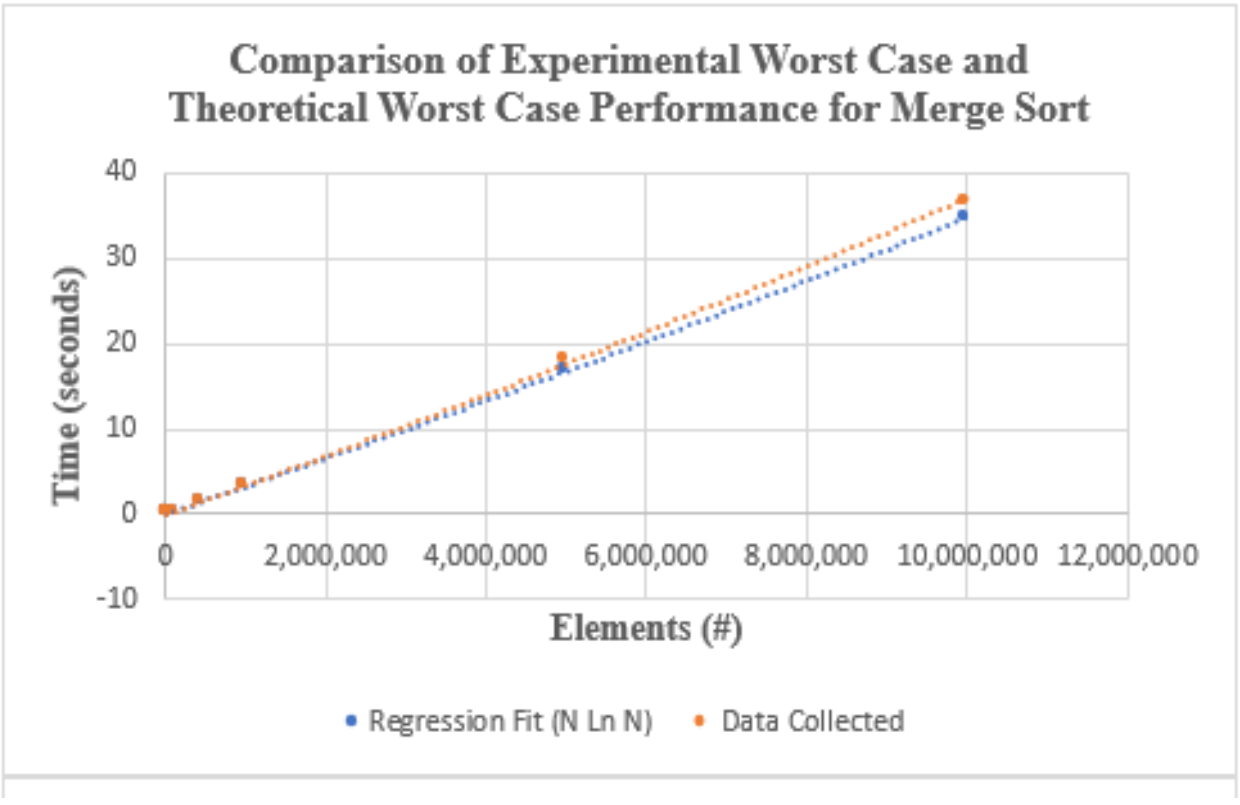
	Median Run Times (seconds)				Standard Error				% Efficient		
	New	Merge	Parallel	Bucket	New	Merge	Parallel	Bucket	Merge	Parallel	Bucket
1,000	0.5	0.001	0.25	0.2	0.002	0.008	0.014	0.012	-99.8	-50.0	-60.0
10,000	0.7	0.02	0.352	0.22	0.004	0.021	0.007	0.010	-97.1	-49.7	-68.6
50,000	0.67	0.15	0.48	0.34	0.006	0.035	0.009	0.010	-77.6	-28.4	-49.3
100,000	0.73	0.29	0.52	0.45	0.007	0.017	0.020	0.015	-60.3	-28.8	-38.4
500,000	0.89	1.13	0.68	0.58	0.008	0.013	0.034	0.044	27.0	-23.6	-34.8
1,000,000	1.03	3.76	2.59	1.76	0.007	0.015	0.042	0.007	265.0	151.5	70.9
5,000,000	1.98	16.64	10.54	20.5	0.009	0.153	0.062	0.231	740.4	432.3	935.4
10,000,000	3.21	34.78	28.12	140.89	0.010	0.142	0.204	0.358	983.5	776.0	4289.1

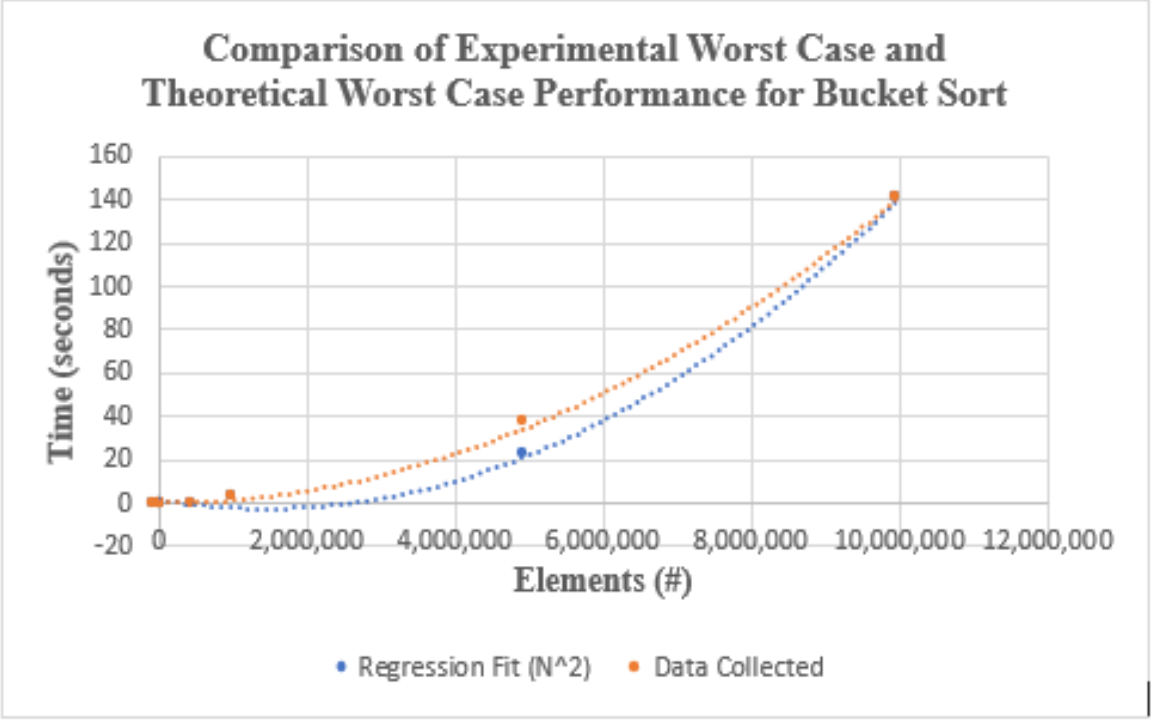



Comparison of run time of various sorting algorithms
Run Time (seconds)
160.000
140.000
120.000
100.000
80.000
60.000
40.000
20.000
0.000
0
2,000,000
4,000,000
6,000,000
8,000,000
10,000,000
Data Size (#)
Bucket Sort
Merge Sort
Parallel Sort
New Sort


Comparison of Experimental Worst Case and Theoretical Worst Case Performance for Parallel Sort
TIme (seconds)
30
25
20
15
10
5
0
-5
0
2,000,000
4,000,000
6,000,000
8,000,000
10,000,000
12,000,000
Elements (#)
Data Collected
Regression Fit (N Ln N)


Comparison of Experimental Worst Case and Theoretical Worst Case Performance for Stat Sort
Time (seconds)
4
3.5
3
2.5
2
1.5
1
0.5
0
0
2,000,000
4,000,000
6,000,000
8,000,000
10,000,000
12,000,000
Elements (#)
Data Collected
Regression Fit (N)


Comparison of Experimental Worst Case and Theoretical Worst Case Performance for Merge Sort
Time (seconds)
40
30
20
10
0
-10
0
2,000,000
4,000,000
6,000,000
8,000,000
10,000,000
12,000,000
Elements (#)
Regression Fit (N Ln N)
Data Collected



Discussion:

Data was collected through a total of 20 trials for each data size, ranging from 1000 elements to 10000000 elements. The averages of these trials were taken and analyzed through InStat. The means of these trials were graphed and a regression from Excel was derived. The data points fit a linear regression with an R^2 of 0.997. It can be inferred from this that the run time of this algorithm is indeed O(n). To confirm that these data points accurately reflected the time complexities of these different algorithms, the run times of sorting these same data sets using merge sort, bucket sort, and parallel sort were collected and graphed. The data points of these respective sorting algorithm fit their respective run times - n log n for merge sort and parallel sort and n^2 for bucket sort. It could then be inferred that these regressional fits did indeed reflect the time complexities of these algorithms. These regressions were then graphed

against each other to show the respective run times of these algorithms and it is evident that Stat sort works at a much more efficient rate as the number of data elements increases when compared to the other algorithms. To further ensure that these regressions did fit their respective time complexities, chi-squared goodness-of-fit tests were conducted in Mathematica. The respective p-values were found; these p-values were all larger than the level of significance of 0.05. This signifies that the regressions from the chi-squared goodness-of-fit test did not differ much from the expected time complexities of these given algorithms.

The efficiency of sorting these data sets was also calculated for each data size and each sorting algorithm. As seen in the first table, it is evident that Stat sort actually takes a longer time to sort these data sets when compared to merge, parallel, and bucket sort until upto data sets reaching 500000 elements. However, as the number of data elements increases to 10000000 elements, Stat sort works upto 4000% more efficient than bucket sort and approximately 1000% more efficient than merge sort. Therefore, the initial inefficiency of Stat sort does not present any major drawbacks as data sets - big data sets - will be reaching sizes much larger than 10000000 elements. This further implies that when dealing with small sized data, Stat sort is not the optimal sort to use. Furthermore, seeing that Stat sort is approximately 700% times more efficient than parallel sort, the multithreaded counterpart of merge sort, applying multithreading to Stat sort could alleviate computation power as well as reduce the run time.

Because this initial phase of research involved only using normally distributed data sets of varying sizes, further research must be done to apply this algorithm to various types of distributions. This will be then be embedded into the Apache Spark mainframe to examine its effect on sorting true big data.

References:

Akil, S. G. (2016). Parallel sorting algorithms. *Department of computing and informational sciences*. Retrieved from https://books.google.com/books?hl=en&lr=&id=jhHjBQAAQBAJ&oi=fnd&pg=PP1&dq=sorting+algorithms+used+today+&ots=uUXYnG4HjD&sig=svU7EkFMs6BJ71fHomQFTN9ilzU#v=onepage&q=sorting%20algorithms%20used%20today&f=false

Bozidar, D., & Dobravec, T. (2015). Comparison of parallel sorting algorithms. Retrieved by https://arxiv.org/ftp/arxiv/papers/1511/1511.03404.pdf

Chong, P. K., Meng, S. S., Mohanavelu, Karuppiah, E. K., Nur'Aini, Omar, M. A., & Amirul, M. (2012). Sorting very large text data in multi GPUs. *IEEE.* doi:10.1109/ICCSCE.2012.6487134

Davidson, A., Tarjan, D., Garland, Michael., & Owens, J. D. (2012). Efficient parallel merge sort for fixed and variable length keys. Retrieved from https://pdfs.semanticscholar.org/05d3/72b38bb05c96e7575a9f48fe5e292fa34e0e.pdf

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

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

Hirschberg, D. S. (2013). Fast parallel sorting algorithms. *21. 657-780.* Retrieved by https://arxiv.org/pdf/1206.3511.pdf%20+%20Radix%20sort.%20MI?KA,%20Pavel.%20%5Bonline%5D.%20%5Bcit.%202012-12 30%5D.%20Dostupn?%20z:%20http://www.algoritmy.net/article/109/Radix-sort

Hong, H. (2012). Parallel bucket sorting algorithm. *San Jose State University.* Retrieved from http://www.sjsu.edu/people/robert.chun/courses/cs159/s3/N.pdf

Jocksch, A., Hariri, F., Tran, T. M., Brunner, S., Gheller, C., & Villard, L. (2016). A bucket sort algorithm for the particle-in-cell method on manycore architectures. *International Conference on Parallel Processing and Applied Mathematics.* 43 – 52. doi:10.1007/978-3-319-32149-3_5

Odeh, S., Mwassi, Z., Green, O., & Birk, Y. (2012). Merge path – Parallel merging made simple. *Parallel and Distributed Processing Symposium Workshops & PhD Forum*. doi:10.1109/IPDPSW.2012.202