Soumya Mishra - Creating a new sorting algorithm
Background Research Paper
10/06/2017

Effect of using statistical analysis to sort numbers in the most efficient way possible

Sorting functions are widely used in computer programming. Efficient sorting is important for optimizing the use of other algorithms which require input data to be in sorted lists. They are mostly used in commercial computations, information search on different search engines, operations research, event-driven simulation, numerical computations, combinatorial search, and string processing; and 25% - 50% of all the work performed by computers require sorting data (Akil, 2016). Several sorting algorithms have been created over time to provide better run time performances. 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 different types of sorting algorithms: comparisonbased 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 sorting algorithm used today is the 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.

There is also a new algorithm that allows the distribution of big data across multiple servers on a cluster using Java and Python: Hadoop MapReduce. Hadoop MapReduce is an algorithm that consists of the map job, which splits the data-set into chunks which are processed in a parallel manner (filtering and sorting, essentially) (Wang et al., 2012). The data set can then be analyzed by these map tasks which run in parallel across the Hadoop cluster. MapReduce also consists of a reduce job, which takes the output, stores it in a system, and then takes care of the scheduling tasks (Wang et al., 2012). Before the map job, splitting of the input data occurs through a parallel process (Wang et al., 2012). This will reduce the amount of heap errors in a given program, and can also speed up the program for each server will have to process and analyze a chunk of the data set (Wang et al., 2012).

On a more local scale, multithreading can also be used in Java and Python to decrease the time complexities of the algorithms. Multithreading allows the program to execute many tasks simultaneously, as opposed to waiting for Function A to be completely done before starting Function B. The built in functions of multithreading – new, runnable, running, blocked, dead – allow the programmer to manipulate how the threads interact with each other and when each thread functions in relation to the other threads. So, not only does this decrease the time complexity, but this also eliminates the issue of memory-allocation problems on a smaller scale. Multithreading is mainly used in Java because Java is compatible with the use of multiple threads at once, as opposed to Python. Python is not the ideal language to multithread in because of the nature and use of this language. CPython implementation has a Global Interpreter Lock (GIL) which only allows one thread to be executed at a time; one exception is parallel IO operations because these are not parallel CPU computations (Monaknov, 2016). The multiprocessing module in Python can be used to perform simple computations; however, this requires full use of the CPU (Monaknov, 2016). The number of processes when using the multiprocessing module does not affect the run time significantly (Monaknov, 2016). This is why multithreading is mainly implemented in Java because it uses Python syntax and Java methods (Jython).

However, one useful aspect of python is the implementation of Panda, which is an open source Python library for data analysis. Initially, Python was only used to prep data sets, but not the analyze these data sets; R, SQL, and even Excel were used to analyze the data sets. Panda now allows Python to also be used for data analysis. Two data structures that Panda introduces to Python are series, one dimensional objects, and DataFrame, tablular data structure, which are built on top of NumPy. Given a data set, Panda can figure out the distribution as well as the variation of the data set, which will be used to achieve the most important goal of this research project.

Most of these improvements will be implemented in this research to make bucket and merge sort more efficient while decreasing the space complexity on one processor/GPU.

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.

Figure 1: Brief diagrammatic process of proposed algorithm

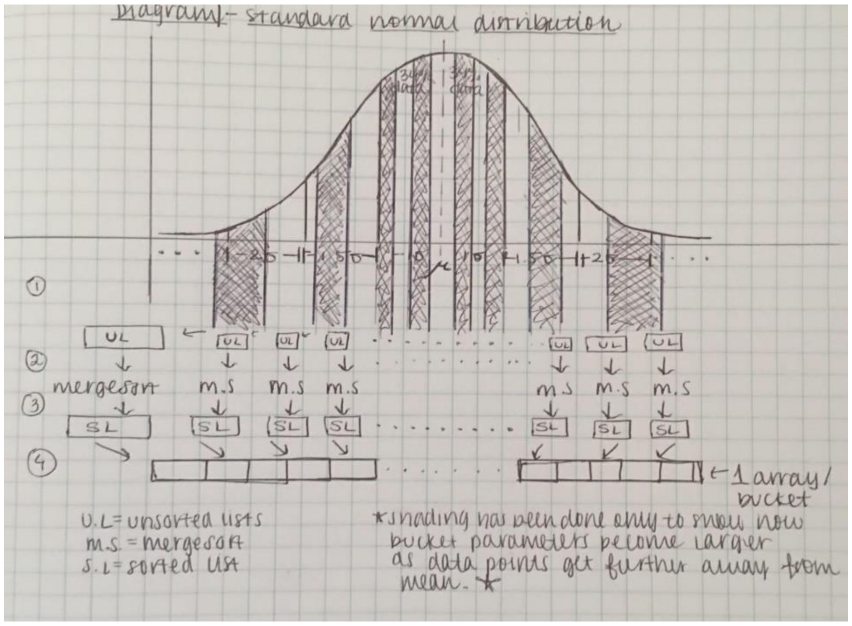
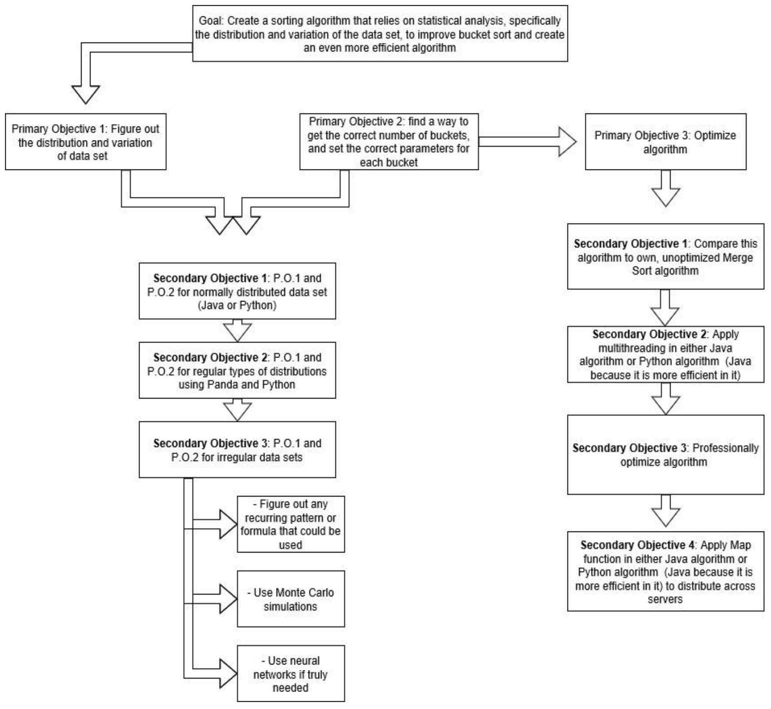


Diagram - standard normal distribution
34% data
34% data
-2σ
-1.5σ
-1σ
μ
1σ
1.5σ
2σ
1
2
3
4
UL
mergesort
m.s
SL
1 array/bucket
U.L = unsorted lists
m.s = mergesort
S.L = sorted list
shading has been done only to show how bucket parameters become larger as data points get further away from mean.

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). After this unoptimized algorithm is created, multithreading, MapReduce, and professional optimization will be incorporated to decrease the time complexity of this algorithm. To ensure that this algorithm is more efficient than leading algorithms, I will implement my algorithm into an algorithm that heavily relies upon sorting algorithms. If it runs faster than before, this will be proof that my algorithm has effectively made sorting much more efficient.

Figure 2: Flowchart of primary and secondary objectives


Goal: Create a sorting algorithm that relies on statistical analysis, specifically the distribution and variation of the data set, to improve bucket sort and create an even more efficient algorithm
Primary Objective 1: Figure out the distribution and variation of data set
Primary Objective 2: find a way to get the correct number of buckets, and set the correct parameters for each bucket
Primary Objective 3: Optimize algorithm
Secondary Objective 1: P.O.1 and P.O.2 for normally distributed data set (Java or Python)
Secondary Objective 2: P.O.1 and P.O.2 for regular types of distributions using Panda and Python
Secondary Objective 3: P.O.1 and P.O.2 for irregular data sets
- Figure out any recurring pattern or formula that could be used
- Use Monte Carlo simulations
- Use neural networks if truly needed
Secondary Objective 1: Compare this algorithm to own, unoptimized Merge Sort algorithm
Secondary Objective 2: Apply multithreading in either Java algorithm or Python algorithm (Java because it is more efficient in it)
Secondary Objective 3: Professionally optimize algorithm
Secondary Objective 4: Apply Map function in either Java algorithm or Python algorithm (Java because it is more efficient in it) to distribute across servers

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 from 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

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

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.2012.09.001