Soumya Mishra - Creating a new sorting algorithm
Background Research + Timeline
09/01/2017

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: 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 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 worst-case 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. 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 parallelizing of the algorithm allows the time complexity to decrease significantly, but also causes the space complexity to increase a little bit. The various studies differed by their methods of using these GPUs to sort the buckets. 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. However, this modified algorithm still has memory-fetch conflicts because more than one processor simultaneously is accessing the same memory location (Hirschberg, 2013). The use of linked-lists, or lists that are connected to their parent or daughter lists or arrays, complicate this problem and even increase the run time by 20% due to memory allocation calls. This wastes unnecessary space and limits the size of the list inputted (Corwin & Logar, 2012). Despite multiple solutions being implemented, there are virtual memory problems.

Hong (2012) addresses the memory-allocation problems presented by Hirschberg (2013) during his research as he uses OpenMP API as 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. Another group of researchers decided to divide the data into multiple buckets and use different GPUs to sort the various buckets using radix sort, which proved to be inefficient because radix sort only converts lexicographical and fixed-length keys, or keys that are all the same bit. The idea to place buckets on various GPUs proved to work faster than the current bucket sort and it was non-dependent on the distribution of the data set (Dehne et al., 2016). However, this uses a lot of expensive technology that would not be feasible.

Furthermore, there have been methods to improve merge sort as well and most of them involve around parallelizing the sorting, 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. Because the blocks are updated based on two moving windows, one block cannot update its register window until everything is sorted and vice versa. This causes a load-balance issue in that if the overlap in the window range is small, some blocks will be used very little or not at all (Davidson et. al, 2012).

A key observation is that none of the sorting 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 groups, which can be sorted separately and then 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.

The first primary objective will be to find a way to figure out the distribution of the data sets. For data sets that fit one of the basic distributions prevalent today (like normal, t, chi-squared), I will use Panda, my first secondary objective, and have it recognize the distribution and figure out the spread of the data set. However, if this proves to increase the time complexity or if a data set with an irregular distribution is given, then I will create a new formula to get a general idea of the data set’s distribution, my second secondary objective. This formula will be based on finding the mean of the data set and using standard deviation – or some statistical element like this – to get the idea. If this still proves to reduce the efficiency of the algorithm, I could implement neural networks to determine the type of distribution. This would be beneficial even when the data set entered does not fit one specific distribution. These networks could be manipulated to combine different types of distributions to understand how the data is spread.

My second primary objective will be to find a way to get the correct number of buckets, and set the correct parameters for each bucket. It is necessary for number of elements in each bucket to be about the same so minimize the time it takes to sort the buckets. 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 bigger and bigger. For example, if a data set is normally distributed, there will be a mean and standard deviation. The algorithm will then use the standard deviation to see where the data concentration is heaviest. Of course, this will be within 1 standard deviation of the mean, where 68% of the data lies. Then, the algorithm will figure out how many data points are within this interval, and divide that number by a thousand, thus giving the number of buckets to be created. To figure out the parameters of the bucket, some function will be needed to be created in which some fraction of the standard deviation times some factor is added or subtracted from the mean and previous parameters. If this proves to reduce the efficiency, neural networks can be used to automatically figure out the bucket parameters and sizes based on the distribution of the data. Thus, my secondary objectives will be to figure out a formula, or multiple formulas based on distribution, that will set the parameters of the buckets and to get the right number of buckets so that no extra space is used.

[insert diagram]

Third, it will be important to optimize the program and ensure that the time complexity is linear. One of my secondary objectives in order to do this will be to reduce the number of passes as much as I can - the number of passes in this algorithm already are significantly less than those of leading algorithms so this will just make my algorithm even more efficient. Due to the complexity of professionally optimizing the algorithm, I will have to code an existing sorting algorithm – like merge sort, bucket sort, insertion sort – and compare it to the run time of my non-optimized algorithm, one of my secondary objectives. Furthermore, I can try to reduce the space complexity by distributing the data set across multiple servers. If this algorithm proves to be more efficient, I will work to optimize it professionally. Lastly, I will then use an algorithm which heavily relies upon sorting algorithms, and implement my algorithm instead of the algorithm initially used in it. If it runs faster than before, then this is proof that my algorithm has effectively made sorting much more efficient.

In order to achieve the primary and secondary objectives listed above, I will have to first learn, up to an intermediate level, Python and Panda, which will go hand in hand. Based on the courses I have started working on, learning these and typing out the framework of my algorithm should be done in two months or less.

Next, I will have to come up with the various formulas to figure out the number of buckets and parameters of the buckets. I will have to collect data points to figure out any pattern and create an equation based on that pattern. This could take me about two to three months for this is the most crucial part of my research.

Then, writing my own merge sort or existent sorting algorithm will not be difficult for I could use code found online and just clean that code up. Comparing this sorting algorithm with mine will take about one month or less because I will have to keep making changes to make mine faster if it is not on the first try. If none of my manual modifications are making it more efficient, then I will have to use neural networks; and in that case, I will need another month or so to learn a good amount about how they work.

Lastly, if I can pass the last step and my algorithm proves to be more efficient, then I can connect professors or professional workers in this field to see if they can help me professionally optimize my program and carry on from there. This would take me the rest of the year and maybe the summer to implement my program in an actual sorting-based algorithm to test whether or not it is truly more efficient.

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