High Performance and Parallel Computing -Individual Project-

Savvas Giortsis

June 19 - 2023

1 The Problem

As computers and technology progress forward, we find ourselves with machines that have access to more and more CPU cores. Sorting algorithms that were previously based on sequential techniques need to be re-adapted to take advantage of this increase in core count. Bucket sort is one such algorithm. Bucket sort is at its core a very simple sorting algorithm that simply divides up all the elements to be sorted into buckets. Those buckets are then merged to create the sorted array. This simplicity often comes with drawbacks, such as spatial complexity and element distribution issues. I choose to further the challenge a bit by implementing this sorting algorithm to work for doubles, both negative and positive. What tends to happen however is that the buckets are indexed in positive integers, thus elements that are negative or elements that are to be placed in buckets over the last bucket (the largest value bucket) end up clustered in the first and last buckets. This is a distribution issue that can also be solved using mathematical techniques, however, this is outside the scope of this assignment which should purely focus on parallelization. Distribution however is not only affected by elements that are out of range, but also with their actual distribution. Sorting elements with different distributions can impact the performance and efficiency of the bucket sort. When the elements are unevenly distributed across buckets, some processors or threads may have a higher workload than others, as threads are given a fixed set of buckets to work on, often leading to load imbalances and reduced parallelism.

2 The Solution

The resulting parallelized bucket sort algorithm is implemented using the OpenMP library for parallelization. Arrays are used rather than structs to increase spatial locality in the caches.

The bucket sort function is, similarly to its sequential counterpart, split into sections that each do one step in the bucket sort algorithm. The first step is to populate the buckets with their respective elements and make sure that those elements within the buckets are sorted. This task can be parallelized quite easily by allowing each thread to work on individual buckets. While having to iterate through the array of elements multiple times in order to find the correct bucket for the corresponding thread. Doing it this way eliminates the need for any synchronization within this step, as it eliminates all critical sections as threads only add and sort their own separate buckets.

After have been sorted within their corresponding buckets, we perform the merge. The merge is the second and last step in the bucket sort algorithm. Merging is done sequentially, as this step requires access to all buckets to recreate the sorted array, which would lead to the need for a critical section if this would be done in parallel. This critical section could be detrimental to performance which is why this section is sequential.

3 Experiments

The results presented in this rapport have been produced with my personal portable computer, the Apple Macbook Pro (2021) with the M1 pro chipset with 10 cores, 8 of which are performance cores and 2 are efficient cores. The computer has 16GB of RAM. The tests were executed on battery power which may impact the performance of the device compared to when being plugged in.

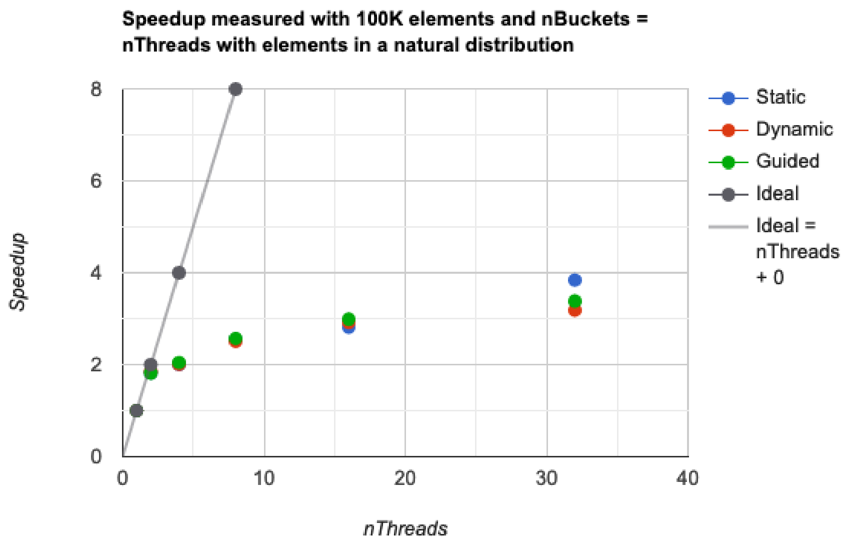

Figure 1: Measured speedup when sorting 100K elements with the number of buckets equal to the number of threads used to conduct the sort. The sorted elements are naturally distributed.

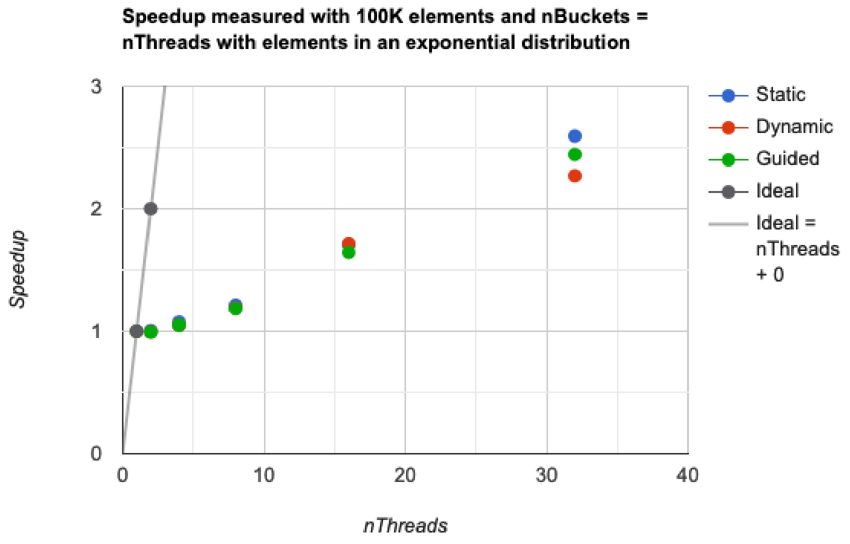

Figure 2: Measured speedup when sorting 100K elements with the number of buckets equal to the number of threads used to conduct the sort. The sorted elements are distributed exponentially.

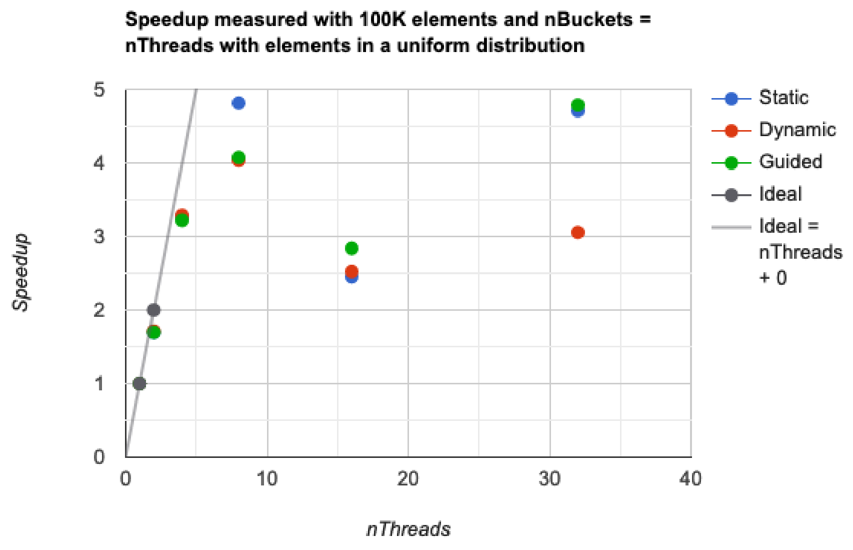

Figure 3: Measured speedup when sorting 100K elements with the number of buckets equal to the number of threads used to conduct the sort. The sorted elements are distributed uniformly.

From the graphs above, 1, 2, and 3, we can observe notable performance differences between using different scheduling options in openMP when using 100000 elements and the number of buckets equal to the number of threads. While the results are relatively similar for a lower thread count, higher thread counts, 16 and 32, see a divergence in performance, with static performing best overall, guided coming a close second, and dynamic performing the worse. When looking at the different element distributions we can clearly see that exponential distributions perform the worse. A uniform element distribution clearly benefits with a much higher speedup than the other two distributions. In Graph 3, we can also observe that 16 threads perform worse than 8 or 32 threads indicating that the Operating System is starting to reschedule the threads to optimize the possible throughput of the algorithm.

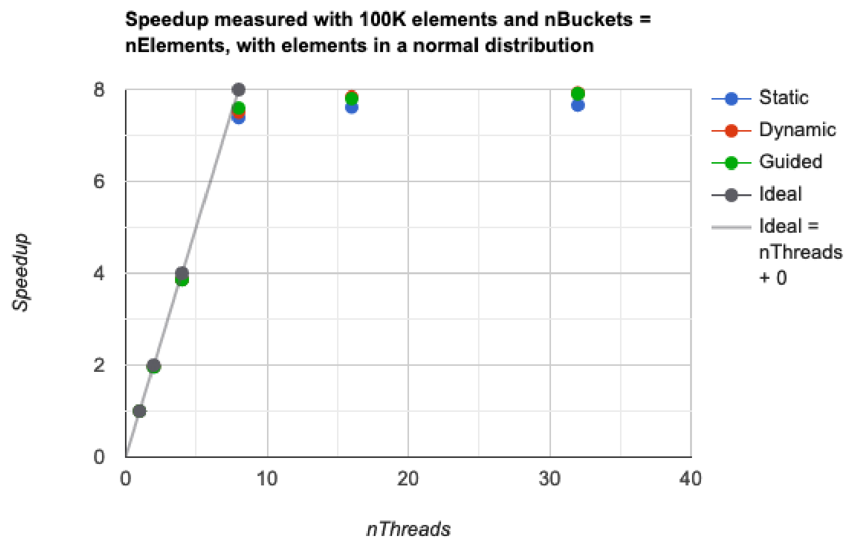

Figure 4: Measured speedup when sorting 100K elements with the number of buckets equal to the number of elements. The sorted elements are naturally distributed.

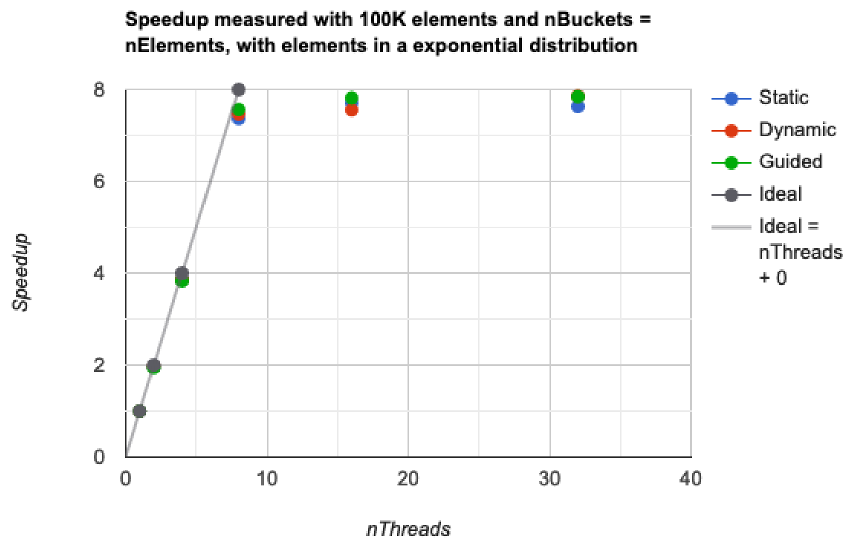

Figure 5: Measured speedup when sorting 100K elements with the number of buckets equal to the number of elements. The sorted elements are distributed exponentially.

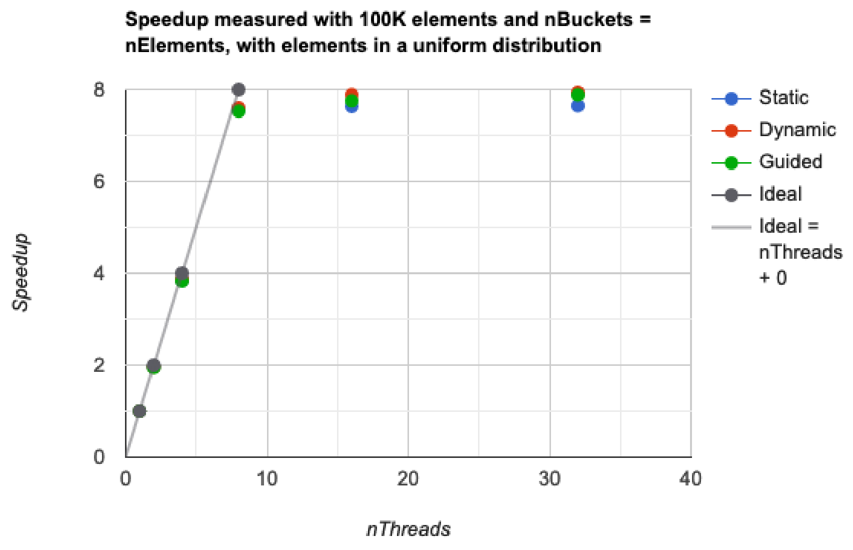

Figure 6: Measured speedup when sorting 100K elements with the number of buckets equal to the number of elements. The sorted elements are distributed uniformly.

In graphs 4, 5, and 6, the number of buckets used instead equals the number of elements rather than the number of threads used to sort the elements. Here we observe very similar performance between all three scheduling options. However, the speedup measured is notably higher compared to using as many buckets as threads. Between the element distributions, there isn't much, if any difference in speedup measured when you have as many buckets as elements.

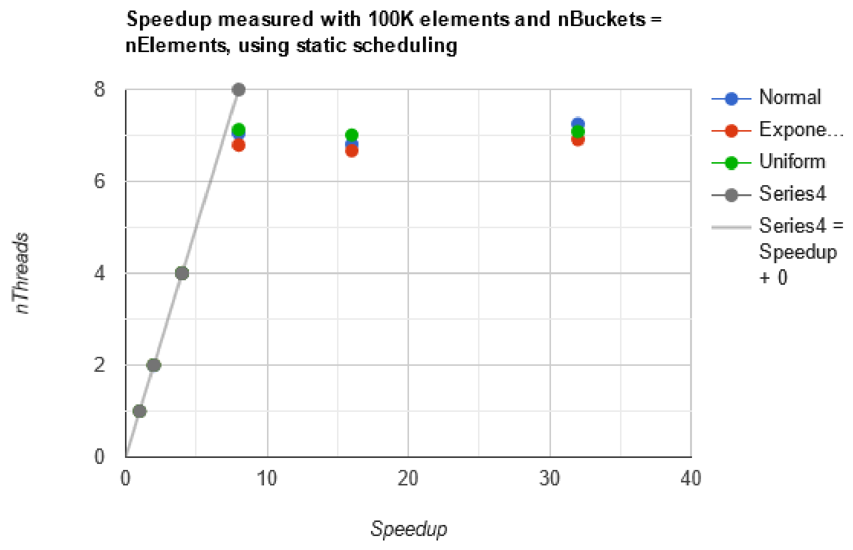

Figure 7: Measured speedup when sorting 100K elements with the number of buckets equal to the number of elements using static scheduling.

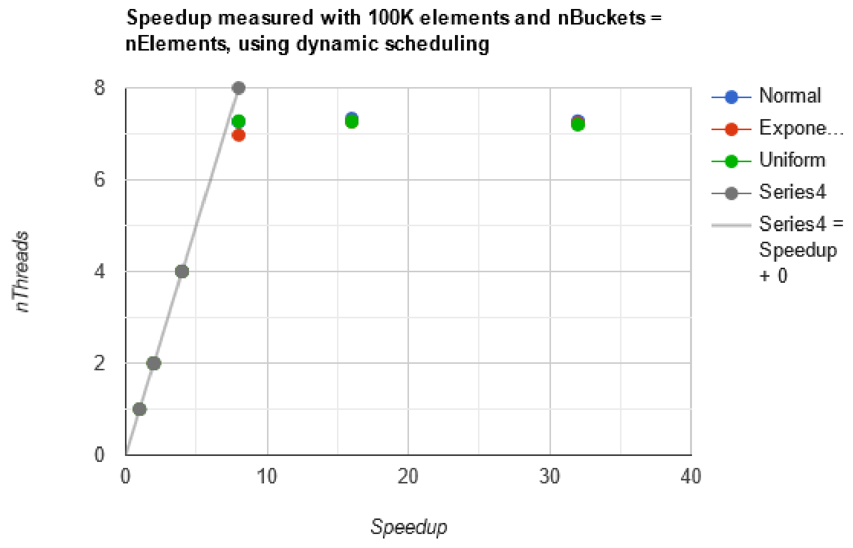

Figure 8: Measured speedup when sorting 100K elements with the number of buckets equal to the number of elements using dynamic scheduling.

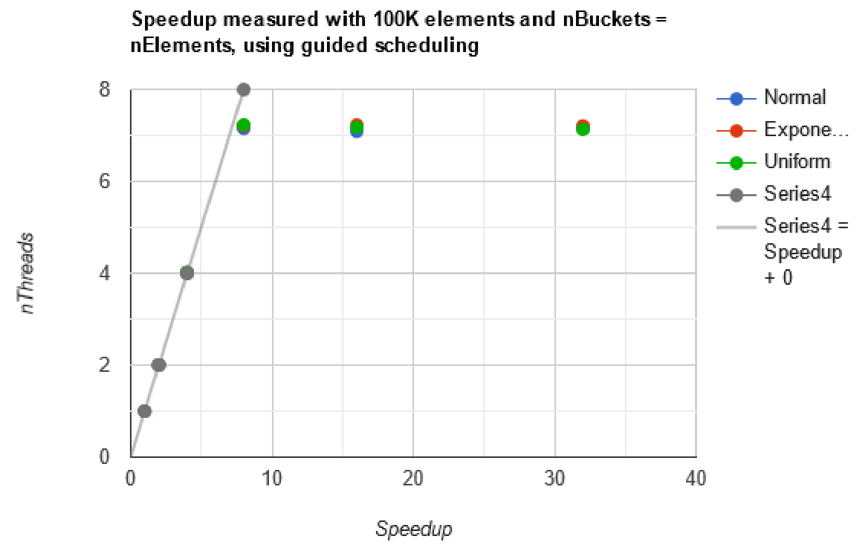

Figure 9: Measured speedup when sorting 100K elements with the number of buckets equal to the number of elements using guided scheduling.

In graphs 7, 8, and 9, we compare the different scheduling options with the different distributions of elements. In all three examples, we cannot with certainty prove that one scheduling option is better than another with a certain element distribution when the number of buckets used is equal to the number of elements that are to be sorted.

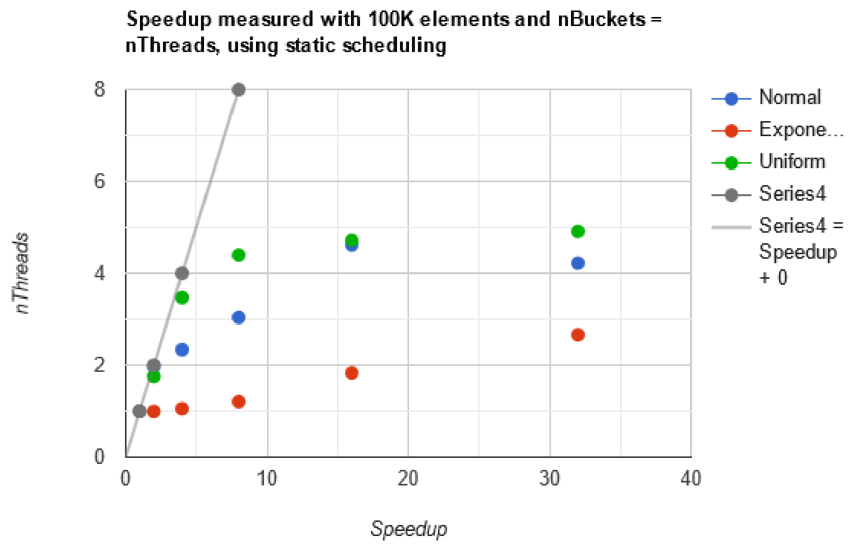

Figure 10: Measured speedup when sorting 100K elements with the number of buckets equal to the number of threads using static scheduling.

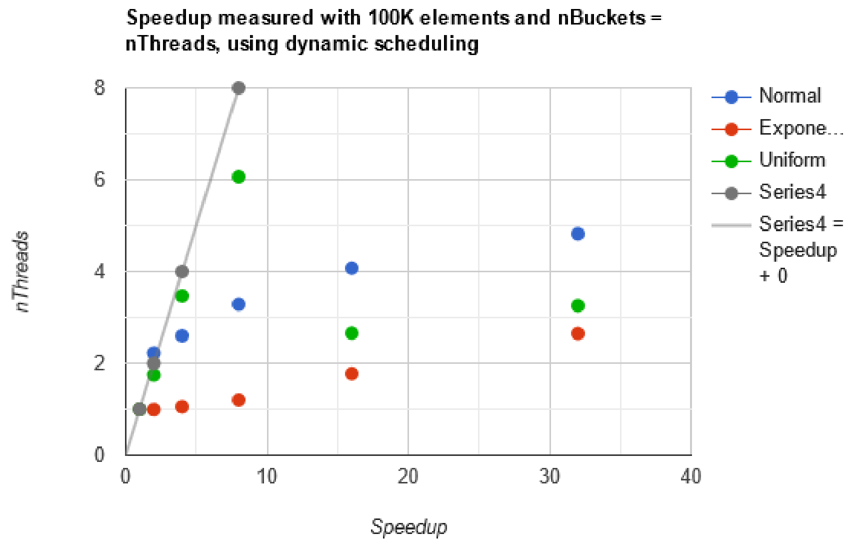

Figure 11: Measured speedup when sorting 100K elements with the number of buckets equal to the number of threads using dynamic scheduling.

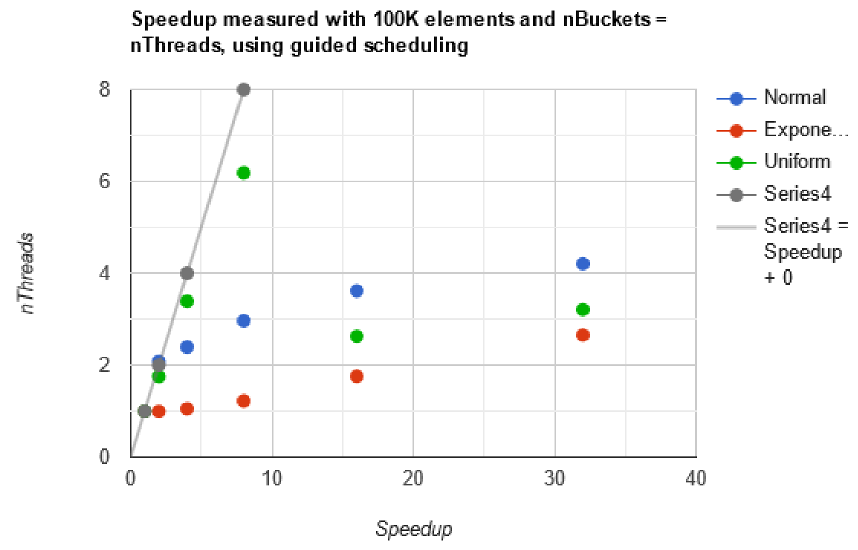

Figure 12: Measured speedup when sorting 100K elements with the number of buckets equal to the number of threads using guided scheduling.

In graphs 10, 11, and 12 we also compare the different scheduling options against each other, with the number of buckets this time equaling the number of threads used. In these instances, there is a clear difference between the three distributions.

In graph 10, using static scheduling, we notice that sorting elements that are distributed uniformly, is much better than sorting elements that are distributed exponentially. However, this is only true as long as the number of threads used is equal to the number of physical cores the machine has access to.

In Graph 11 we observe a similar conclusion as we saw in Graph 10. Uniformly distributed elements vastly outperform any other distribution, in fact, compared to static scheduling, we observe much better performance on 8 cores for the uniform distribution. However, when the number of threads increases past the physical core count, the uniform performance quickly decreases and the normal distribution shows the highest speedup of all other distributions.

In Graph 12, using guided scheduling, the speedups are identical to those observed using dynamic scheduling.

The correctness of the code can be confirmed using the graphical representation that is also implemented in the code. The results of this can be seen in Figure 13.

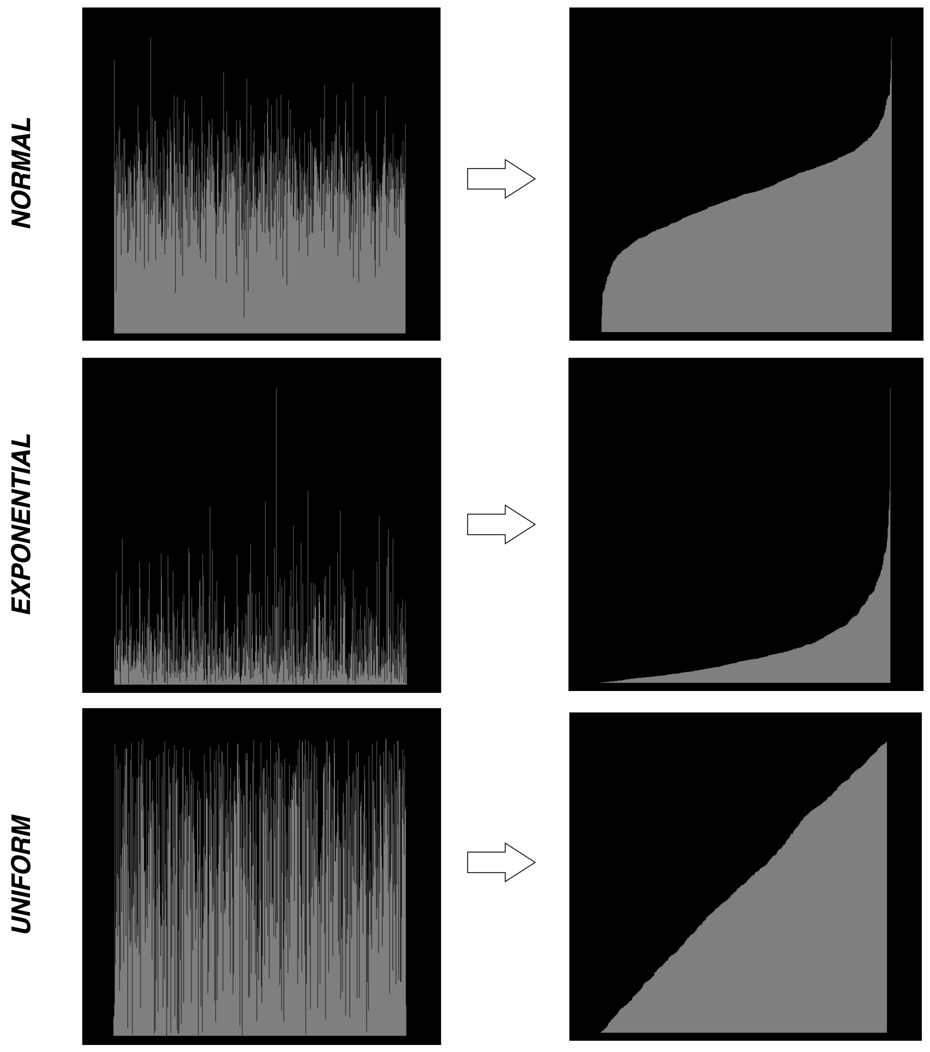

Figure 13: Visual representation of the three different data sets, prior to and after they have been sorted.

Figure 13 visualizes the different required data sets, normal, exponential, and uniform. The visualization shows the datasets prior to them being sorted using the bucket sort algorithm, as well as after they have been sorted. We can also here clearly see the difference between the datasets and their distributions.

3.1 Disclaimer

All results presented in this study are based on the fastest achieved performance of 3 rounds of execution. As described in the Hardware and Specification section, all results are achieved on battery power, which might impact performance. The speedup was calculated using the following formula (1):

$$Total\_Speedup = \frac{Sequential\_Time}{Parallel\_Time} \tag{1}$$

4 Conclusion

In this project, we explored the parallelization of the bucket sort algorithm and transformed it to be able to take advantage of the increasing number of CPU cores in modern machines. Our goal was to optimize the sorting performance by distributing the workload among multiple threads. We implemented the parallel bucket sort using the OpenMP library and compared different ways of optimizing the performance of the algorithm using different scheduling options. We also compared running the sort using a different number of buckets to see what works best and in what situations.

Through the experiments, we can conclude that the sorting algorithm performs best when sorting data that is uniformly distributed when the number of threads is less than or equal to the number of physical threads of the executing machine. We have also witnessed the effects of OS thread rescheduling to maximize the throughput of the algorithm in Graph 3.

We also learned that using as many buckets as there are elements can be beneficial to the observed speedup of the algorithm, however, this can be due to the immense load that is placed on the algorithm when running it sequentially. It would be interesting to further test the algorithm with a number of buckets that is a multiple of the number of threads rather than equal to the number of threads.

Overall the algorithm implemented did not manage to reach the optimal performance, where the speedup is linear to the number of threads used to conduct the sort. However, the speedup is still good with a close-to-linear performance increase of up to 4 threads.

4.1 Potential improvements

I believe that there are several improvements that can be made to the code, one of which is the approach for choosing what bucket the element should go in. Currently, we loop through all the elements for each bucket that is used, in order to pick specific elements just for that bucket. I believe there are other ways of completing the same task with vastly less time, but this might result in the need for other means of synchronization between the threads, such as locks or transactions. This may however slow down the code rather than speed it up, however, this may only apply for datasets that are not of significant size, as the current approach of not using any critical sections is not very scalable.

Another section that could be parallelized is the merging of the buckets in the last stage of the algorithm. Doing this in the time frame of this assignment however proved to be quite intricate, with logic more optimal and easier to implement in higher-level languages. Although I hypothesize that for smaller datasets, the resulting sequential merge performs similarly or better than if we were to parallelize the merge.

5 References

- Wikipedia article on bubble sort - `https://en.wikipedia.org/wiki/Bucket_sort`.
- Bucket sort pseudocode - `https://www.javatpoint.com/bucket-sort`.
- Generating Uniform Distribution - `https://stackoverflow.com/questions/1340729/how-do-you-generate-a-random-double-uniformly-distributed-between-0-and-1-fro`
- Generating Normal Distribution - `https://stackoverflow.com/questions/2325472/generate-random-numbers-following-a-normal-distribution-in-c-c`.
- Generating Exponential Distribution - `https://stackoverflow.com/questions/34558230/generating-random-numbers-of-exponential-distribution`.