High Performance and Parallel Computing
-Individual Project-

Savvas Giortsis

June 4 - 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. This task can be parallelized quite easily by just splitting up the original array that we wish to sort, into different threads that each place elements into buckets. This however can induce data races, as multiple threads might want to add elements to the same bucket simultaneously, which can easily be fixed by stating that this is a critical section using the `#pragma omp critical` directive.

Critical directives can however be detrimental to performance as they are mutually exclusive if not named. Naming such a critical section could be tricky, as ideally, we want one name per bucket as we do not want to block all threads when just one of the buckets is being added to. Instead, we want to just block the specific bucket we want to add to. This would allow threads to keep adding to other buckets while we block threads wanting to add to our bucket when performing the add. One way to avoid this effect is to have private buckets. By creating private buckets we skip the critical sections when adding elements to the

buckets. However, after all, threads have finished adding to their corresponding private buckets, we are required to perform a merge step. In the merge step, we merge all private buckets into their corresponding public buckets. This adds another step in the algorithm which in turn slows it down. We explore the runtime effect of this technique on the bucket sort algorithm and compare it to the `#pragma omp critical` directive in the Experiments section.

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.

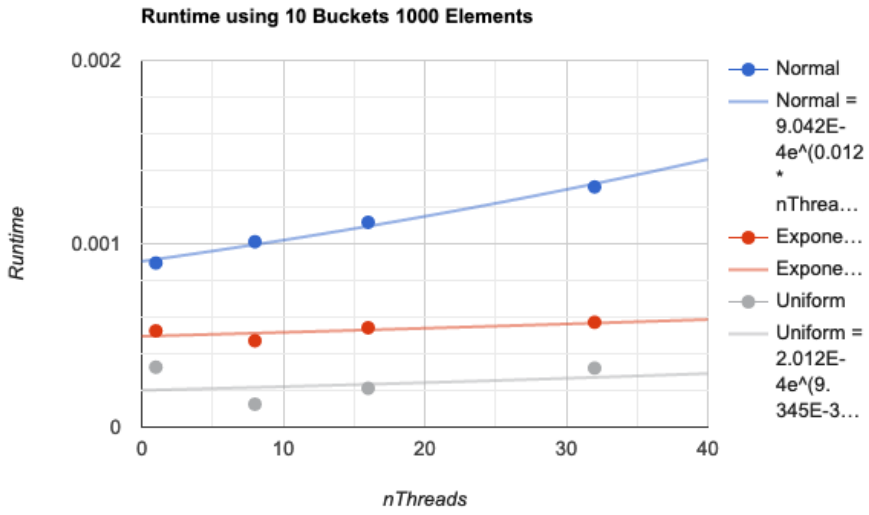

Figure 1: Time is taken to sort 1000 elements with 10 buckets using the `#pragma omp critical` directive.

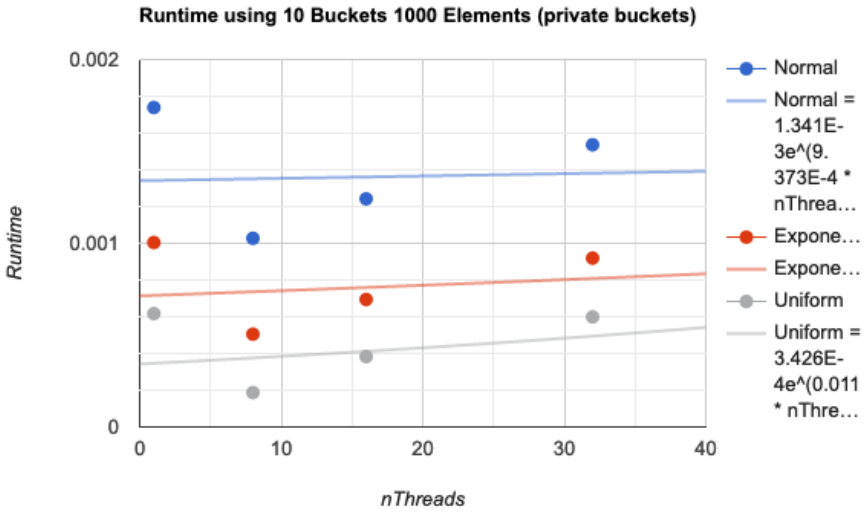

Figure 2: Time is taken to sort 1000 elements with 10 buckets using the private buckets approach.

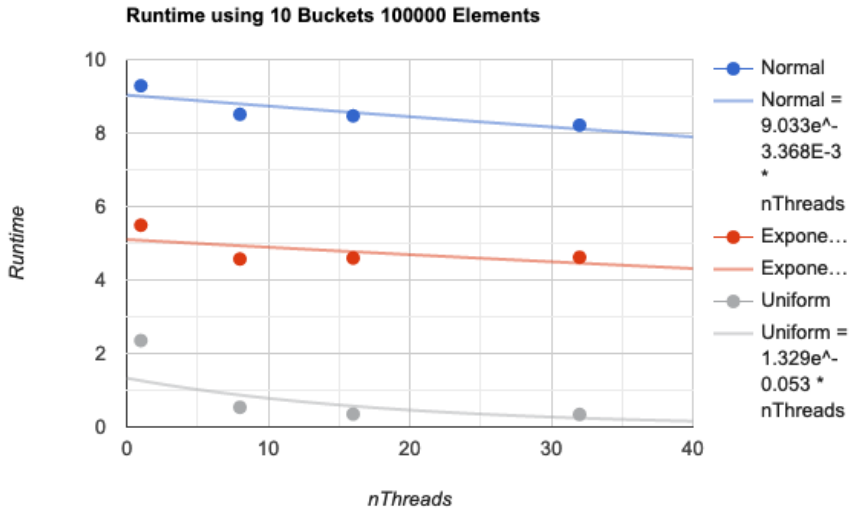

Figure 3: Time is taken to sort 100000 elements with 10 buckets using the `#pragma omp critical` directive.

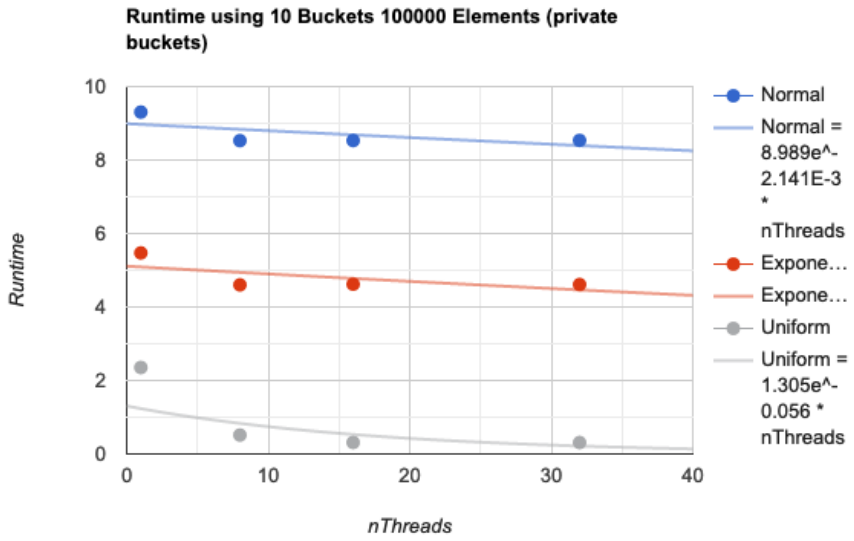

Figure 4: Time is taken to sort 100000 elements with 10 buckets using the private buckets approach.

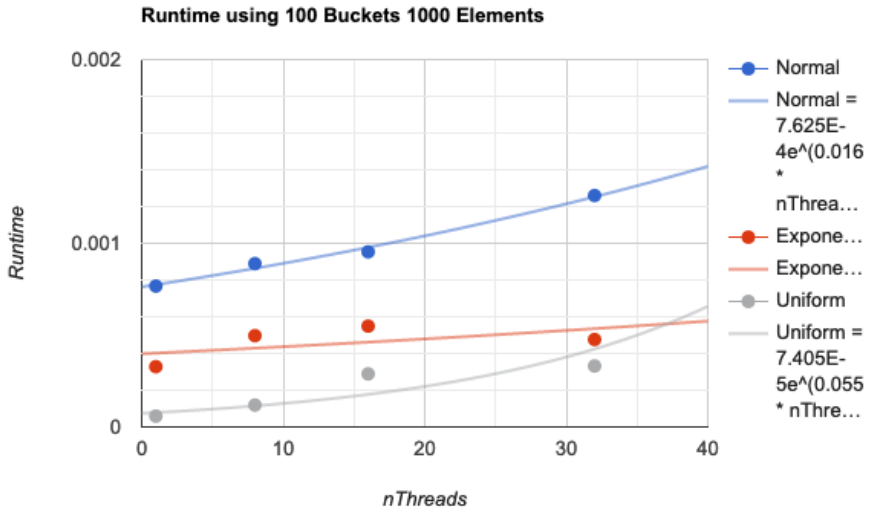

Figure 5: Time is taken to sort 1000 elements with 100 buckets using the `#pragma omp critical` directive.

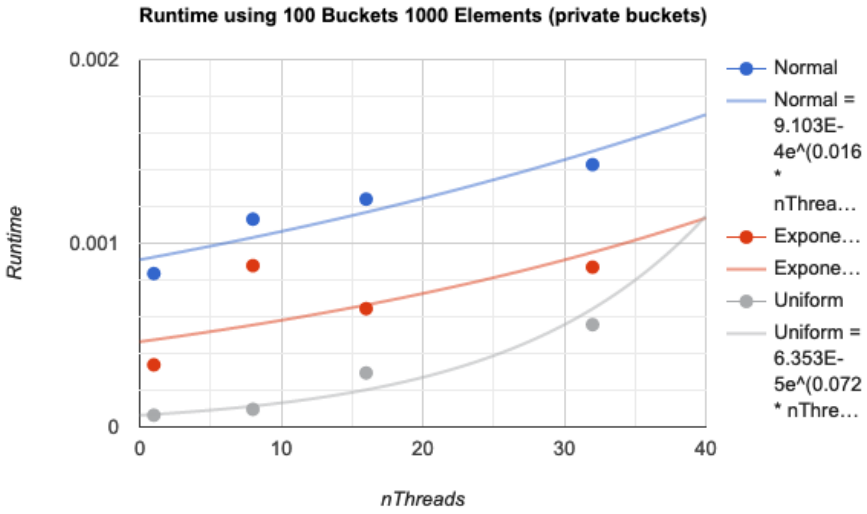

Figure 6: Time is taken to sort 1000 elements with 10 buckets using the private buckets approach.

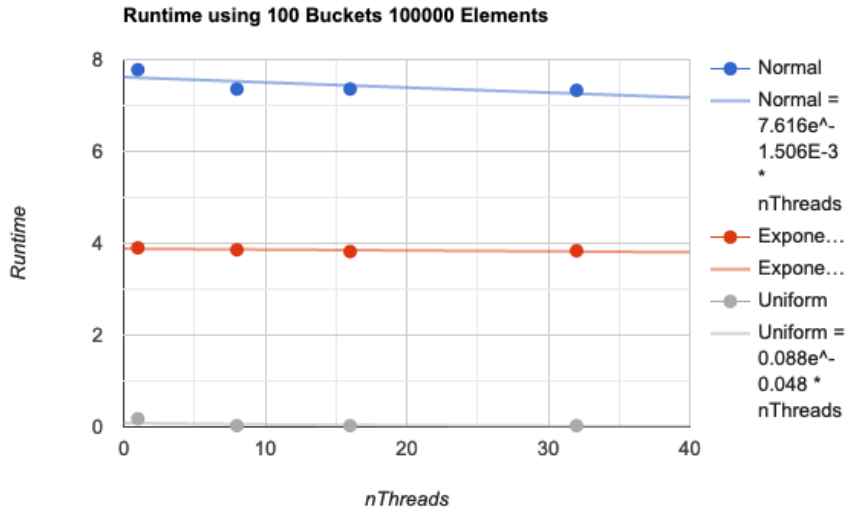

Figure 7: Time is taken to sort 100000 elements with 100 buckets using the `#pragma omp critical` directive.

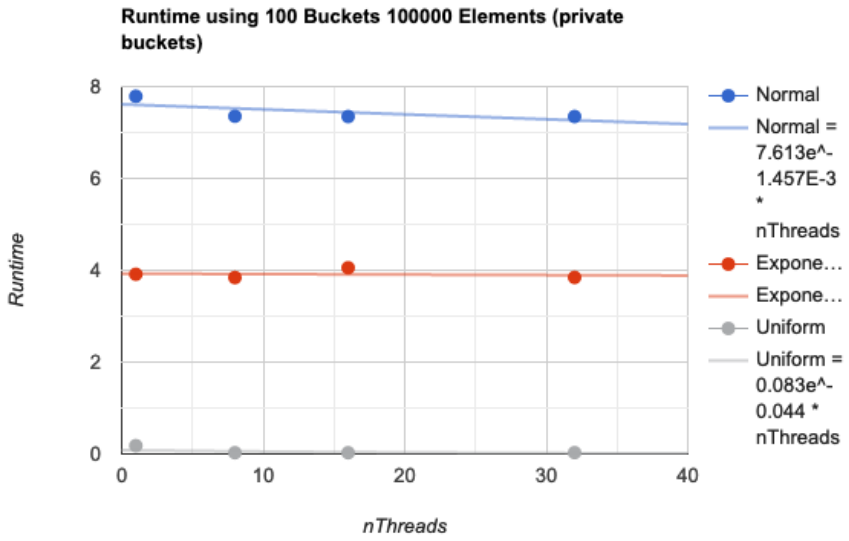

Figure 8: Time is taken to sort 100000 elements with 100 buckets using the private buckets approach.

From the graphs above, 1 through 8, we can deduct any notable performance gains when going from using the `#pragma omp critical` directive to using private buckets. The only noticeable difference in these graphs is between Figures 1 and 2, where it is clear that the private buckets approach is slower for the small dataset (N = 1000, nBuckets = 10) when running it sequentially on one thread. For larger datasets or even a larger number of buckets, this anomaly goes away. It seems that either openMP is able to optimize the lock used when defining the `#pragma omp critical` directive or that the merging of the private buckets takes a big toll on the performance, as there too, synchronization is required, although not nearly as much. In Figure 9, we can see that the performance between the two really is identical. When it comes to parallel performance, we can see that there are improvements for certain data-set distributions when running the sort sequentially compared to running it using 8 threads. We can deduce that running the sort concurrently for normally distributed data sets is least optimal, as the graphs only display minor improvements from 1 to 8 threads, or even a longer run-time compared to running it sequentially. The most optimal distribution is the uniform distribution, which manages to be sorted the fastest.

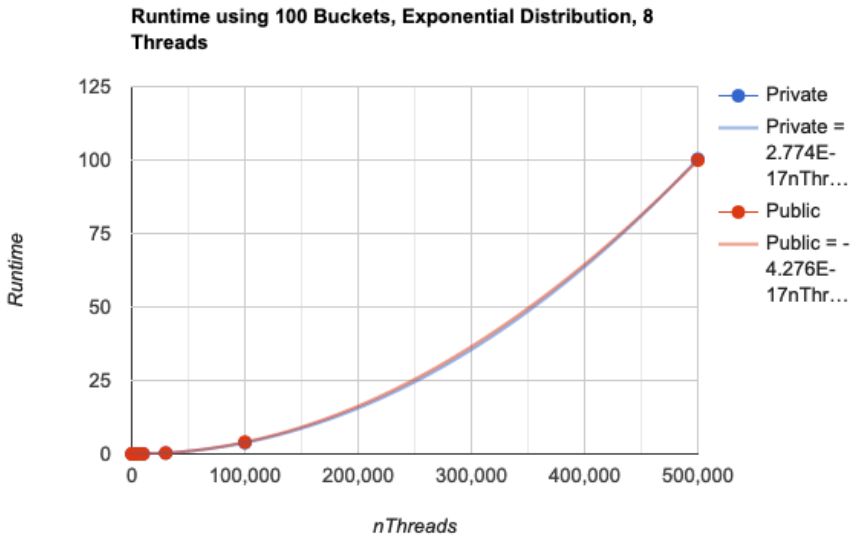

Figure 9: Time is taken to sort using 100 buckets, 8 threads, and an exponential data set of different sizes. Comparison between the two approaches.

The speedup of the two different versions can be seen in Figures 10 and 11. We can clearly see that there is a clear speedup on all cases of around 1.15, however, there is no notable speedup for the normal and exponential distributions when increasing the number of threads past 2. The uniform distribution however shows a good, but not ideal speedup when increasing the number of working threads. We also compare the two versions' speedups from the uniform execution in Figure 12, where we see again that the difference between the two versions really is not notable.

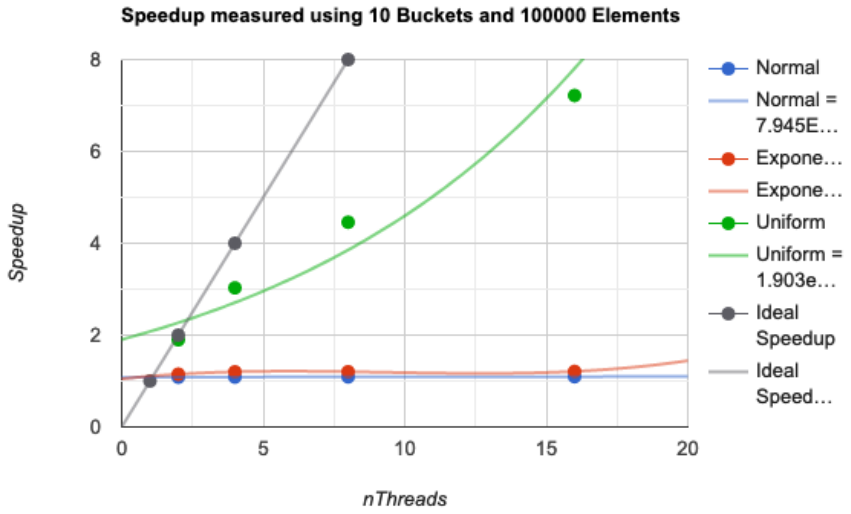

Figure 10: Speedup observed when using 10 buckets, 100000 elements on a uniformly distributed data set for the critical section implementation.

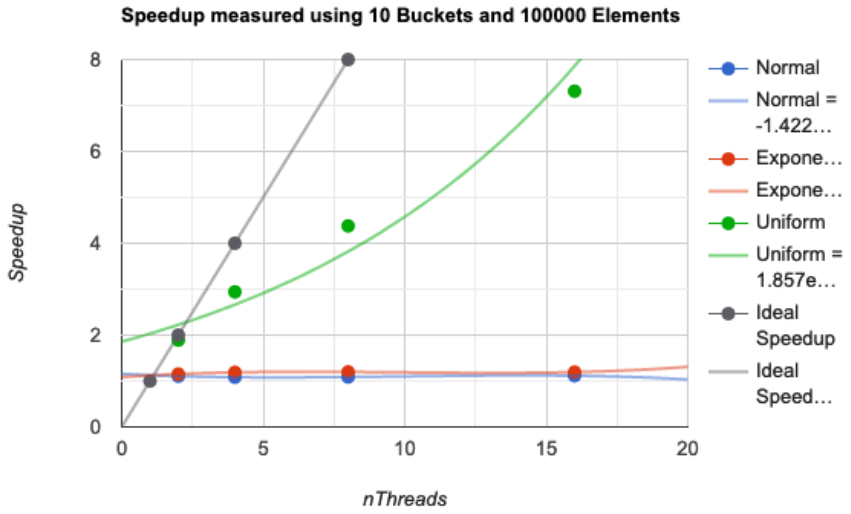

Figure 11: Speedup observed when using 10 buckets, 100000 elements on a uniformly distributed data set for the private bucket implementation.

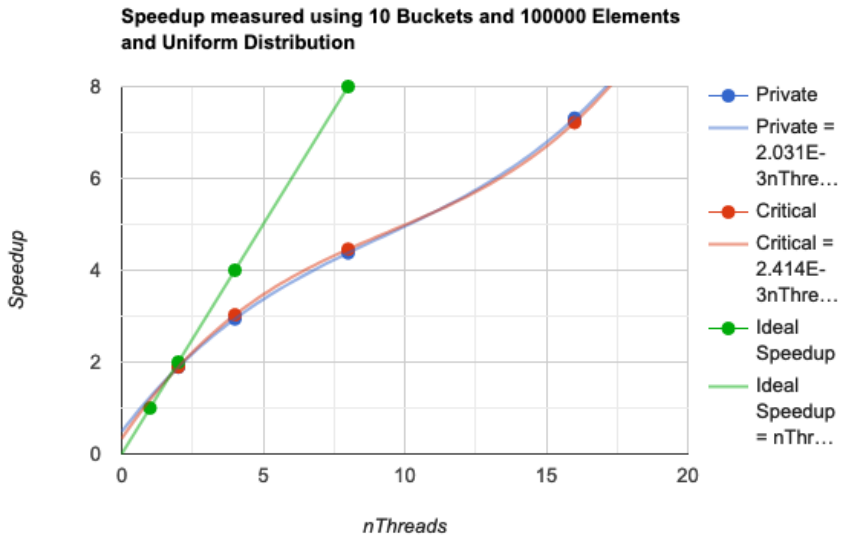

Figure 12: Comparison of the speedup between the two implementations when using 10 buckets, 100000 elements on a uniformly distributed data set.

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.

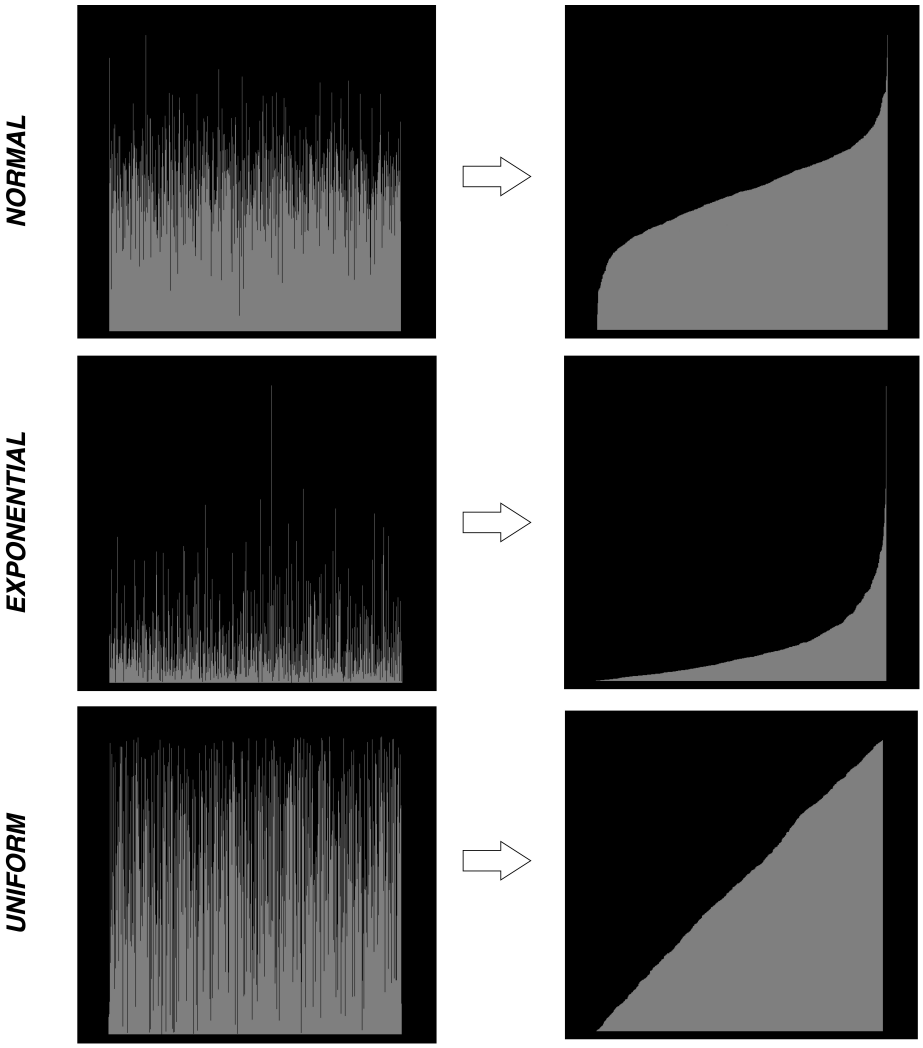

Figure 13: Visual representation of the three different data sets, prior to and after they have been sorted.

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 two different approaches, one using the `#pragma omp critical` directive and one utilizing private buckets to eliminate critical sections in the bucket-filling stage of the sorting algorithm.

Through the experiments, we can conclude that the use of private buckets does not consistently provide significant performance gains compared to the `#pragma omp critical` directive. In fact, for smaller data sets when executed sequentially, the private buckets approach exhibits slower performance. One possible reason for the similarity of the performance is the overhead of merging the private buckets or behind-the-scenes optimizations of the `#pragma omp critical` directive in compile time.

Regarding parallel performance, the benefits of parallel execution were limited for certain dataset distributions. Generally, the uniform distribution showed the best performance improvement when sorting concurrently. However, for other distributions, the performance gain was minor or even resulted in longer runtimes compared to executing the sort sequentially.

Speedup was also not great when sorting normally and exponentially distributed data sets. However when sorting datasets of uniform distribution the speedup observed is good, but not optimal (linear to the number of threads).

4.1 Potential improvements

I believe that there are several improvements that can be made to the code, one of which is load balancing. Currently, the load balancing can be improved, instead of using static scheduling, dynamic scheduling could help improve performance when the load balance is off between threads.

Another improvement would be to improve the merging efficiency for the private buckets approach. Currently, this section requires a critical section that is guarded by a `#pragma omp atomic` directive. However, due to time constraints, I was unable to come up with a better approach for this project.

When sorting the buckets, currently bubble sort is used. This is yet another area of improvement, where we could use a more efficient sorting algorithm, or even a dynamic scheme, where bubble sort is used for smaller buckets and some other algorithm, perhaps quick sort, is used for buckets with a larger number of elements.

Yet 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`.