# COL380 Assignment 1

Sayam Sethi

January 2022

## Contents

## 1 Implementation

The following steps give an idea of the implementation and design choices:

1. Merge sort was used for `SequentialSort`.
2. `SequentialSort` was called if $n/p < p$ since $p^2$ pseudo splitters cannot be selected in this case.
3. The first $n \mod p$ buckets contain $n/p + 1$ elements and the remaining buckets contain $n/p$ elements.
4. $A_i$'s are not constructed explicitly in the code and $R$ is generated directly from $A$.
5. $R$ is sorted and the array $S$ is considered implicitly as the elements of $R$ at suitable indices.
6. Using tasks, the array $A$ was split into almost equal $p$ partitions and a task was scheduled for each partition. The bin for each element was computed (using binary search) and the number of elements in each bin for each thread.
7. The counts for each bin across different threads were summed up.
8. Using tasks, $p$ arrays $B_i$'s were created for each bin of length computed via the counts in the previous step. Each task copied corresponding elements from $A$.
9. Prefix sum of counts was computed.
10. Using tasks, $B_i$'s were copied back to $A$ at the correct locations and the corresponding subset was sorted by `SequentialSort` or `ParallelSort` depending on the size.

## 2 Time and Space Complexity

### 2.1 Time Complexity

The average case time complexity of `ParallelSort` is given as:

$$\begin{aligned}&O(p^2) + O(p^2 \log p) + O\left(n + \frac{n \log p}{t}\right) + O(p \cdot t) + O\left(\frac{p \cdot n}{t}\right) + O\left(\frac{p \cdot (n/p + n/p \log(n/p))}{t}\right)\\ =&O\left(p^2(1 + \log p) + n + pt + \frac{(p+1)n + n \log n}{t}\right)\end{aligned} \tag{1}$$

The assumption made in the above analysis is that the bins are almost euqally split such that no bin has a size larger than the threshold. Also, $t$ above is the number of threads.

### 2.2 Space Complexity

The average case space complexity is given as:

$$O(p^2) + O(p \cdot t) + O(n) + O(p) + O(n) + O(n) = O(p^2 + pt + n) \tag{2}$$

Again, the same assumption is made that `SequentialSort` is called for each bin and thus the space complexity of the function call is $O(n)$ and the remaining space is used for the arrays declared in the `ParallelSort` function.

## 3 Parallelism and Scalability

From the time complexity, it is clear that the entire function cannot be parallelised and an $O(n)$ term remains outside the parallel execution. Additionally, the parallel execution involves an $O(pn)$ term (when updating $B_i$'s) which isn't truly parallel. Although, the same loop could have been converted to an $O(n)$ non-parallel loop, however, the same would involve significant cache misses due to the random nature of the bins of consecutive elements.

Following are the plots for different values of $n$ and $p$ and comments for the same are followed:

### 3.1 Graphs

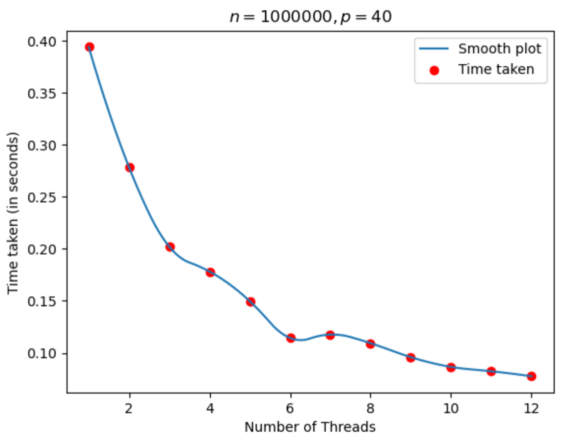

Figure 1: $n = 10^6, p = 40$

The scalability for this case is 5.09× for 12 cores. There is linear decrease until 3 cores and the absolute value of the slope decreases after that. The value of slope decreases even further from 7 cores (and remains almost constant from 6 to 7 cores).

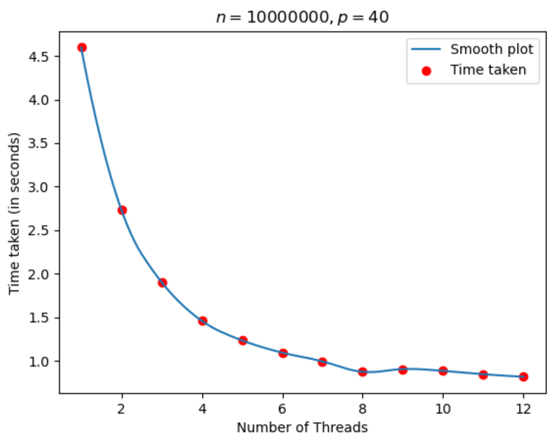

Figure 2: $n = 10^7, p = 40$

The scalability for this is 5.63× for 12 cores. The curve follows an asymptotic pattern and the running time almost saturates by 8 cores. The scalability upto 8 cores is 5.25×. The constancy is partly due to the fact that when scheduling tasks, the division cannot be done equally across all threads and some threads work on fewer loops than the other threads.

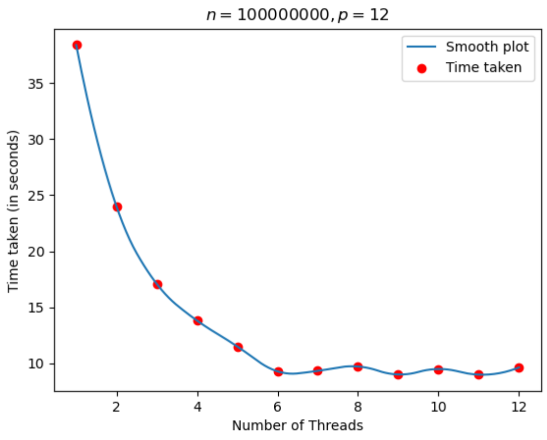

Figure 3: $n = 10^8, p = 12$

The maximum scalability for this is $4.27\times$ for 11 cores. This is much lesser than the scalability in the previous parts. This is due to the fact that the number of bins is fewer and thus the probability of equal division of work reduces. The running time saturates by 6 cores and the scalability then is 4.13. Additionally, the fluctuations in running time for $8, 10, 12$ threads is due to the scheduling scenario where some threads need to perform comparatively more tasks. This increases the execution time.

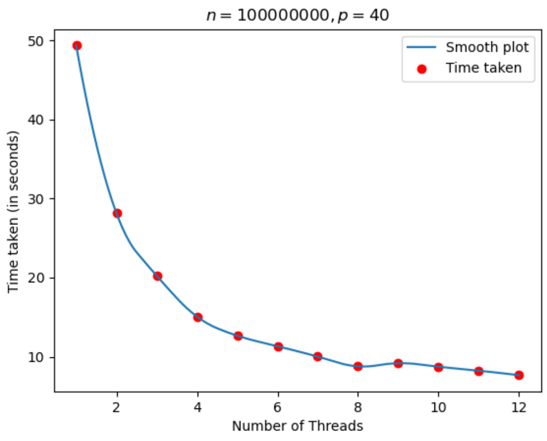

Figure 4: $n = 10^8, p = 40$

The scalability is $6.43\times$ for 12 cores. This situation doesn't have the limitation of reaching saturation like the previous graph. Therefore, the speedup slows down towards 12 cores but there is still a linear improvement from 9 to 12 cores.

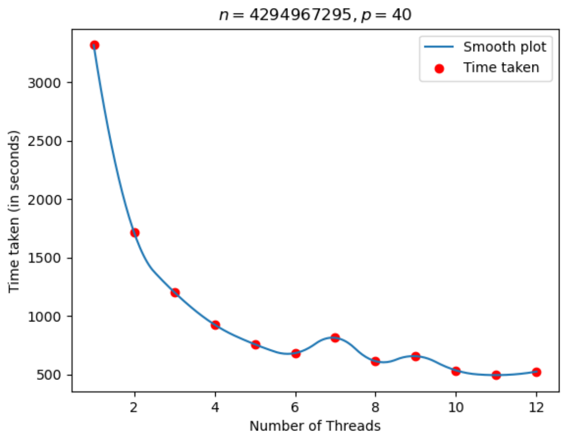

Figure 5: $n = 2^{32} - 1, p = 40$

The maximum scalability achieved is 6.70× for 11 cores. There is significant fluctuation towards large number of cores, however, the general trend is that the running time is significantly reducing even at 12 cores. The same isn't visible in the graph due to the large amount of time taken for fewer number of cores.

## 3.2 Conclusion

The maximum speedup obtained in the entire analysis was 6.70×. The best efficiency of the parallelisation was 0.56. The speed-up would have been better if the analysis could have been performed upto 24 cores however, due to limited resources on HPC, the same could not be done. Overall, every possible loop in the function `ParallelSort` was parallelised and any further optimisation would be micro-optimisations, especially in memory allocation.

**Note:** In the entire discussion above, the number of cores is the same as the number of threads.