Assignment 1, COL380

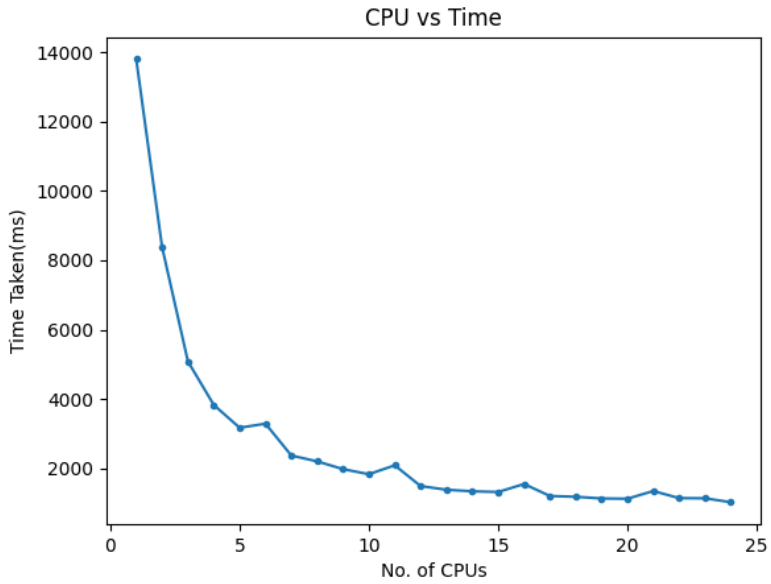
Sarthak Singla, 2019CS10397

Outline of Implementation & Design Choices

- First, if there is only 1 bucket or the number of pseudo-splitters is half of size of array, Sequential Sort is done. This is because with p<=40, we will prefer sequential sort over parallel sort for arrays of size <= 3200 to avoid unnecessary parallelisation overheads.
- Otherwise, first an array containing the pseudo-splitters R is created by looping p times over the contiguous n/p sized buckets of data.
- R is sequentially sorted as its size is small.
- We create an array S containing p-1 equally spaced splitters from R. This has been done because these elements are required for read-only later and bringing them closer fits them in less number of cache lines.
- A 2D array B to store partitions and an array C to store size of partitions is declared. C is stored with padding 64 to avoid false sharing during updates.
- After this, we iterate over data to find size of each partition B0, B1, …, Bp-1. This has been done to know how much space each partition would require. This has been parallelised using OpenMP tasks, as the p different sections can be identified independently. A private variable has been used for storing the counts, and a shared variable is updated only at the end, to avoid the problems of false sharing. The array B is also initialised using the counts. Taskwait is called at end before executing anything that uses a shared variable.
- Prefix sum of array C is done.
- Again, OpenMP tasks are used to fill the array B with the elements of their respective partitions. Finally, based on number of elements and threshold, the partition Bi is either sorted by SequentialSort or ParallelSort. Again Taskwait is called at the end.

- Tasks are also used to transfer each partition Bi into the array data at respective positions, determined from their counts in array C. Each task handles one partition. At the end taskwait is called.
- Finally we delete the arrays allocated.
- For SequentialSort, we use MergeSort for array size above 50 and InsertionSort otherwise, the constant chosen by testing. This is because InsertionSort has smaller constants.
- Except for small loops of size p, we have kept everything parallelised, so that the code is able to scale on increasing number of CPUs and data size.

Graphs



The above graph shows the run for n = 2^25 and p=24. The number of CPUs and threads varies from 1 to 24. We can see a good scalability to multiple cores, with 13-14x scalability in the above example for 24 cores. A sharp decrease in time for 2 CPUs is observed.
Some small variations are visible, but these may be attributed to some other tasks running at the same time on the PC. Also, these might point out to some scope of improving parallelisation, as the spikes seem to occur around the same points across multiple runs.

Similar graphs were obtained with same configuration except varied p.