COL380 Assignment 1: Sorting with OpenMP Tasks

By Mohit Sharma(2019CS10372)

Time(ms) vs Number of cores [For input with n=2^32-1, p=40]

Time(ms) vs Number of cores [For input with n=2^24, p=40]

**Old Report Contents:**

The algorithm followed is as given in the assignment pdf.

After selecting the p-1 elements for the partition range, tasks are used to partition the given array.

The given array is divided in continuous p blocks of size n/p +-1 each. Each block is then processed by different thread to count the number of elements in the in each partition. A 2-d array structure is used to store these counts where count[i][j] represent the count of elements that belong to ith partition in the jth block of array.

A task wait is used to stop main thread from executing further before completion of this counting process. Next to partition array elements I analyzed two choices:

1) Pre-calculate a prefix sum of counts of elements in each partition by every block of array.

(i.e., a 2-D array structure, pref[i][j] representing count of elements in ith partion in first j-I blocks of array). Then another p tasks are made similar to earlier ones where the store the result of portioning of array in a temporary array of same size as our data array which is later copied back to original array.

2) The second choice was to directly do the shuffling of the array in-place without any tasks.

Choice (1) is used because here we use the threads to do the tasks and does not let the resources gets wasted as in case of choice (2) which saves the overhead of memory allocation but then wastes the n-1 threads/cores capacity during that time (assuming number of cores>=number of threads). Hence to scale well in a better hardware available to us it is best to utilize its resources without increasing much of the total computation done.

Letter the partitioned array is recursively sorted by sequential or parallel sort based on the comparison with given threshold value. This recursive work is also done by threads.

Following graph between number of CPU cores(x-axis) vs time ms(y-axis) shows the scalability of the algorithm on large input data array of size (2^24) where 40 buckets are used for partitioning the array and number of cores were increased to 12cores. (For final code submitted (using choice 1 above)).

The scalability achieved with 12 cores is around 7 times as compared to using 1 core only.

**Extra:**

1) This for the Choice (2) mentioned above.

Following graph between number of CPU cores(x-axis) vs time ms(y-axis) shows the scalability of the algorithm on large input data array of size (2^32) where 40 buckets are used for partitioning the array and the number of cores were increased to 24 cores. (This was done using choice 2 above).

From the graph it can be seen that the scalability of the code is not good enough. Initially with increase of cores the time decreases but at after number of cores goes above 8 it saturates and the decrease in time is much less. This happens because during the time main thread shuffles the array for portioning, other threads are just useless and we lose to scale up performance.

For this same algorithm the runtime for data of size 2^24 with 40 buckets and 12 cores is as follows: The saturation after 4 cores is clearly visible here as the dataset is small.

(Evaluated runtimes for this algorithm to compare with the algo where choice 1 is followed, but due to long queues could not do it for a greater number of cores).

**High Memory But Core Utilization:**

I tried another algorithm in which I created copy of the given data files(basically p new array for each partition) at each recursive step of parallelSort call. Here I created the partition in each task and does not need to wait for other tasks to finish before using the copy of the partion to continue sorting the already partitioned part in recursive calls. I tought this will help in increasing scalability as when more cores will be available the threads wont need to wait for each other and would be busy most of the time. But it just not only increased the computation needed to be done by each thread but also showed the bad affect of time consumed in memory allocation, as memory was allocated recusively hence and was not freed until end of sorting of all partitions hence the memory allocations further down the recursive stacks take even more time. With increase in cores more and more threads actively created copies for partition and hence the expected performance scalability was nullified.