Report for Assignment 1, COL380

Shrey Patel

January 2022

Contents

1 Implementation and Design Decisions

1.1 Sequential

1. **Obtaining pseudo-splitters(Steps 1+2:**
   I have not explicitly divided the given array into buckets to obtain the set of pseudo-splitters R. Instead, I have taken the first p elements from each of the p buckets implicitly by keeping track of the boundaries of each of the buckets. The first $n\%p$ buckets will have $\frac{n}{p}+1$ total elements while the rest of the $p-(n\%p)$ buckets contain $\frac{n}{p}$ total elements. So, the boundaries of i^{th} bin can be obtained as:

   $$A_i = \begin{cases} [i*\left(\frac{n}{p}+1\right),(\text{i+1})*\left(\frac{n}{p}+1\right)) & \text{if i} < n\%p \\ [(\text{n\%p})*\left(\frac{n}{p}+1\right) + \text{i}*\left(\frac{n}{p}\right),(n\%p)*\left(\frac{n}{p}+1\right) + (\text{i+1})*\left(\frac{n}{p}\right)) & \text{otherwise} \end{cases}$$

   Then, R can be obtained as:
   $R[i] = A_{i/p}[i\%p], \forall\ 0 \leq \text{i} < p^2$

   Time complexity: *O(p^2)*

2. **Obtaining partition boundaries(Step 3:**
   The partition boundaries, stored in S, have been calculated using the given method:
   $S[i] = R[(i+1)*p], \forall\ 0 \leq \text{i} < p-2$

   Time complexity: *O(p)*

3. **Calculating accumulated sum i.e. rangecount**
   rangecount is an auxiliary array which helps in rearrangement of elements of original array so that it is partitioned into bins. This process has been described later in the task-based section. rangecount[i] denotes the number of elements in input which occur before the bin B_i (not including itself). It can be calculated from another auxiliary array *count*, which we have defined later.

   $$\textit{rangecount[i]} = \begin{cases} 0 & i = 0 \\ \textit{rangecount}[i-1] + \textit{count}[i-1] & 0 < i < p \end{cases}$$

   Time complexity: *O(p)*

Note that all of the above sequential operations could have been done in parallel using tasks as well. In that case, if we use generate p different tasks to be run in parallel by the threads available, then we would expect a complexity improvement of the factor of p.

But that is not the case since we also need to consider the overhead added due to task scheduling and synchronisation. Here, the number of buckets(p) are much less than the size of the input, so much so that all the above operations can be considered to be taking *O(1)* i.e. constant time with respect to other operations which scale with the input size. Moreover, the overhead mentioned above outweighs the amount of gain which might have obtained by parallelization of these operations. So, making them sequential and executing them in a single thread is more efficient than using tasks to parallelize them.

And for this same reason, we also directly use sequential sort, which I have implemented using quick sort) to sort the input array if the input size is close to the value of p. Particularly, we switch to sequential sort if $n \leq p^2$. So,

ParallelSort(data,n,p) = SequentialSort(data,n) if $n \leq p^2$.

Time complexity: *O(n log n) = O(p^2 log p)* $\approx$ *O(1)*

1.2 Task Based

1. **Dividing input array into bins**

   - After determining the partitioning boundaries S, we have to divide the original array into bins B_i such that $\forall\ b \in B_i$, $S[i-1] < b \leq S[i]$. But since we have to do the final sorting in-place, it would be better if we can just rearrange the elements of the original array so that they are implicitly divided into bins with the above property, instead of allocating a separate array to each bin. And each bin shall be handled by a separate task.
   - So, to facilitate this, we need a pair of boundaries allotted to each task so that they can operate on that separate region on the same shared array. For that, we use two auxiliary arrays *count* and *rangecount* which respectively store the count and base of the bins.
     count[i]: Number of elements in bin B_i
     rangecount[i]: Number of elements which occur before the bin B_i
   - We require counts of every bin before we can calculate rangecounts. Normally, in sequential execution, we would traverse the input array p times, to calculate the counts of each of the bins, one pass for each bin. But instead we can divide this computation into tasks. A bin B_i is allotted to a task T_i such that:

     Define T_i **as**: Traverse the input array, and if some element $a \in B_i$, then increment count[i].
   - So, now instead of *O(n*p)* total time, each task takes *O(n)* time and assuming that roughly all the tasks are executed in parallel in some constant number of rounds, the total time complexity will also be *O(n)*.

2. **Allotting bins to the input elements:**

   - Assuming that we have calculated *rangecounts* using the sequential strategy mentioned earlier, we need to rearrange the elements into their bins. We need to read the input array and write the elements into their appropriate bins. But since multiple tasks will be reading the same input array, writing into the same input array would result into race condition. So, we define a global array B of the same size as input data array. All the tasks will write onto B in their respective bins.
   - Like earlier, each bin B_i can be allotted to a task T_i, where

     Define T_i **as**: Maintain a local counter(initialised to 0). Traverse the input array, and if some element $a \in B_i$, then B[rangecount[i] + counter] = a and increment the counter.
   - Since the nature of the task is the same as in previous point i.e. traversing the input array, the total time complexity will be *O(n)*

   Both the steps above together constitute step 4 of the algorithm.

3. **Sorting the bins**

   - Once again, we allot a bin B_i to a task T_i such that

     Define T_i **as**: If count[i] < threshold, then sort B_i using quick sort. Else, recursively use parallel sort on B_i.

4. **Merging the bins**

   - Here, we can see the advantage of using an entire auxiliary array B, and the exclusive division of bins. If we allocate a separate array to each bin, then we would have to explicitly merge them at this step. But since we have already recursively sorted the auxiliary array B in-place, all we need to do is to copy the auxiliary array B into our input array.
   - This can also be divided into tasks so that:

     Define T_i **as**:
     Copy B[rangecount[i],rangecount[i]+count[i]) to data[rangecount[i],rangecount[i]+count[i])
   - The size of a bin is $\frac{n}{p}$ on average. So the time complexity of this part is O($\frac{n}{p}$) $\approx$ O(n).

None of the above parallel tasks result in race conditions since each bin has its own allocated region in both the input array and the auxiliary array B. Particularly, the task T_i working on bin B_i is limited to the indices [rangecount[i],rangecount[i]+count[i]). And because of the way *rangecount* is calculated, all these partitions are disjoint, which means that each task computes on disjoint sections of the shared array.

Another thing to note is the improvement in the execution time. The division of the computation into tasks once again achieves an improvement by a factor of p, as argued in the sequential section. But, here the original time complexity of each task is *O(n)*, which was *O(1)* in sequential parts. So, a p factor difference here results in a greater absolute difference (i.e. n times) in execution times. And the gain here outweighs the overhead involved in task scheduling and synchronisation. This is the reason I have used tasks in steps 4-6, and not in 1-3.

2 Task Synchronisation

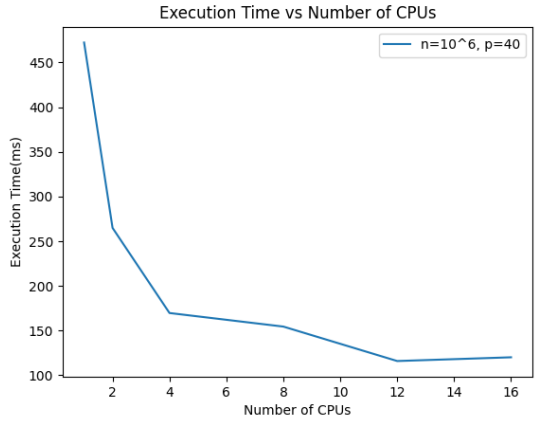
- The first task segment derives the boundaries of each of the bins, and the future tasks use these boundaries to avoid data races. So, the completion of all the tasks in this task segment is a pre-requisite for the correctness of the other steps. So, synchronisation using *taskwait* is necessary after 1.
- Now, theoretically, once the size of the bins are decided, each task can work on its own bin, without affecting the array locations outside its boundaries. So, all the operations on a bin i.e. bin allocation + sorting bin + copying bins to original array can be allocated to a single task without waiting for other tasks before the final synchronisation.
- But, since the number of tasks and therefore the number of bins are greater than the number of threads available, only some tasks can be active out the task pool. And if we combine all the steps mentioned above in a single task, the time taken for a thread to complete the task would increase, and so the tasks which haven't been allocated to any thread are "starved". And larger is the computation inside a single task, larger will be the wait time of unassigned tasks.
- Additionally, since a task in itself is sequential, increasing the length of a task is basically increasing the sequential part. So, in order to increase parallelism, I have separated the above three operations into three different task segments and *taskwait* after each of the segments.
- However, the above reason is speculative.

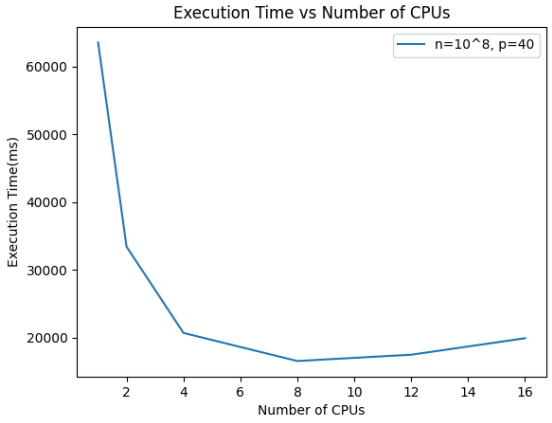
3 Analysis(Execution Time(ms) vs Number of Threads)

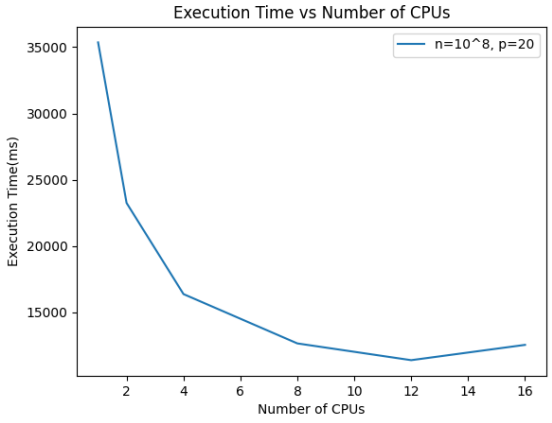
3.1 Outputs:

Threads	n=10^6, p=40	n=10^8, p=40	n=10^8, p=20
1	472.284	63537.5	35362.2
2	264.88	33408.7	23258.8
4	169.678	20674.8	16387.7
8	154.499	16528.8	12669.4
12	115.966	17459.9	11406.2
16	120.143	19891.3	12560.1

3.2 Graphs:







3.3 Observations:

- In each of the three plots, we can see that the improvement in time complexity saturates after a certain number of cores (in our case, 8 or 12). This shows the Amdahl's Law, that some amount of computation is inherently sequential and therefore there is a limit upto which a program can be parallelised.
- The scalability values of the three cases are:
  1. n=10^6, p=40: 4.07
  2. n=10^8, p=40: 3.84
  3. n=10^8, p=20: 3.10

  We can see that for lesser number of buckets for the same value of n, although the total execution time is lesser, the scalability is also worse, as compared to using greater number of buckets.
- The decrease in execution time due to the decrease in the number of buffers may be due to that "starvation" of tasks discussed earlier. So, if there are greater number of buckets for the same number of threads, the number of parallel "rounds" required for the threads to complete all the tasks also increases.
- Now, for different values of n, we can see from the slopes of the graphs that the curve with the larger value of n has greater initial slope as compared to the one with smaller n value. This might suggest that the gain in efficiency can be better observed in larger data. Although, the scalability here is better for lesser value of n.
- One anomalous behavior, common to all the three plots, is the sudden increase in execution time at 16 threads. This might be related to the fact that 16 threads is the maximum that my machine supports. But the exact reason is not clear.