INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 1

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL380 | Prof. Subodh Kumar

Due January 31, 2022

Chapter 1

Parallel Sort Algorithm

I used OpenMP *tasks* to implement parallel sort algorithm as required by the assignment. The algorithm followed and threshold values used, were as specified in the document. Here, I list my other design choices:

1. I used *Quick Sort* algorithm as the sequential sorting algorithm. Since, the generated data is *random*, I refrained from randomising my implementation, and using unnecessary header files. In general, this should not affect scalability of my code. However, in some particular cases (data is already sorted), the code can be slowed down, as I use *taskwait* (see below).
2. At the beginning of ParallelSort(), I check the size of array (n). If $n < p^2$, then I sort the array sequentially and return. Otherwise, I proceed as described in the document.
3. I sort the pseudo-splitters using SequentialSort(), rather than ParallelSort(), as $p^2 \leq 1600$.
4. I have used a *for*-loop that generates p OpenMP tasks. Each task is associated with a particular partition. The task scans the entire array once and stores elements belonging to its partition into its own dynamic array (shared variable: array of arrays).
5. The dynamic array is user-implemented, and it follows an *exponential*-growth strategy to achieve an $O(1)$ amortized time to insert an element into the array.
6. After creating partitions, the same task compares the size of partition with the threshold value ($\frac{2n}{p}$), and sorts as specified in the document.
7. To wait for all the tasks, *taskwait* has been used after the *for*-loop. Once, all *child*-tasks are complete, the function concatenates the different partitions in a sequential manner (into *data* array). I tried implementing this step using more tasks, but the overhead was non-trivial for high values of p. Hence, I finally implemented this step without using tasks.

Space Complexity: $O(n)$, using self-implemented structure

Observations:
I fixed the value of p to be 24, and varied the size of array (n), and the number of CPU cores (and consequently number of threads). I varied n in the list $\{10^5, 10^6, 10^7, 10^8, 10^9\}$, and varied number of CPU cores in the list $\{1, 2, 4, 8, 12, 16, 20, 24\}$. I plot the *Runtime v/s Number oF CPUs* graph for every value of n, to understand the parallelism and scalability of my code. In all figures (1.1 to 1.5), number of threads used is equal to the number of CPU cores.

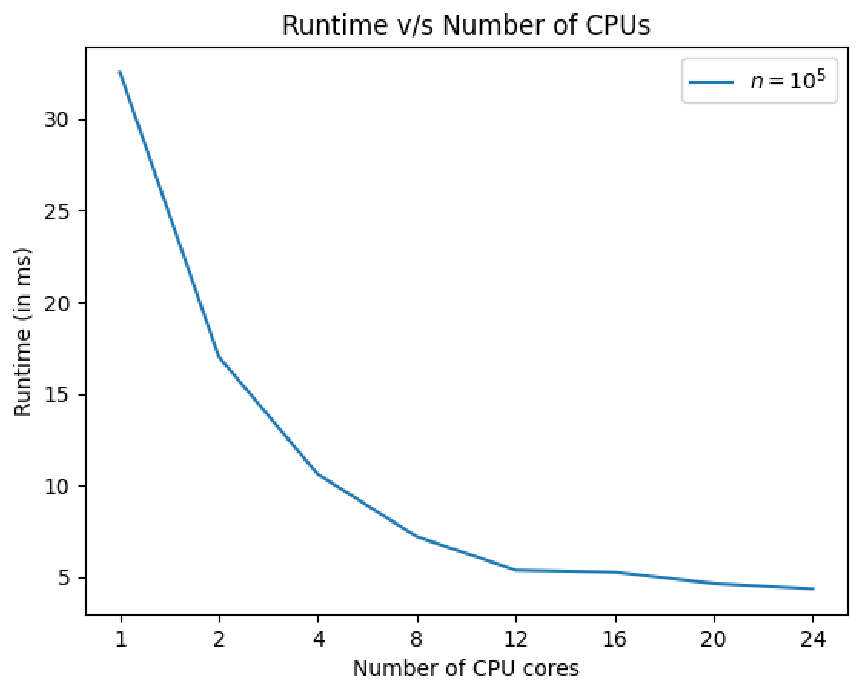

Figure 1.1: $n = 10^5$

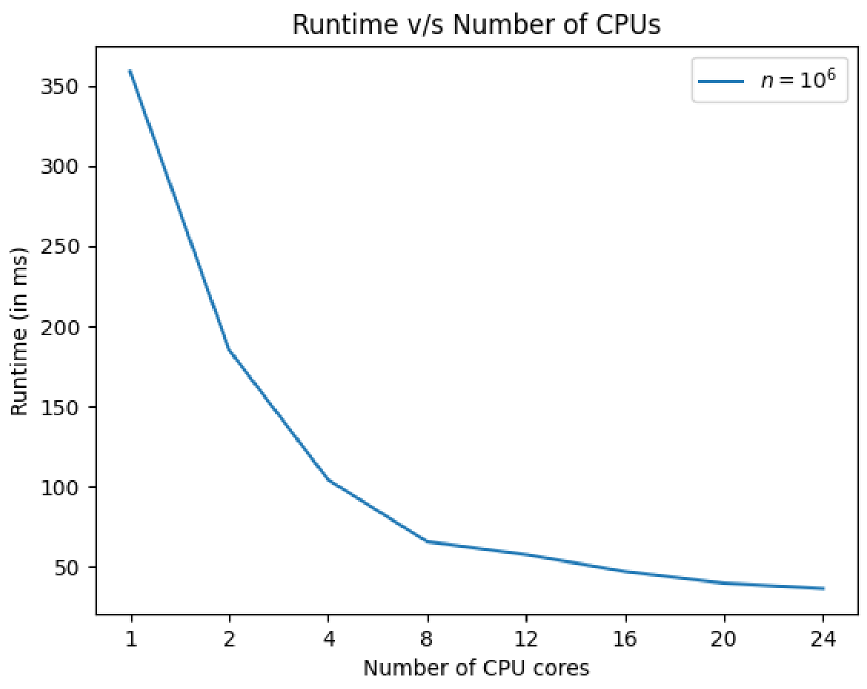

Figure 1.2: $n = 10^6$

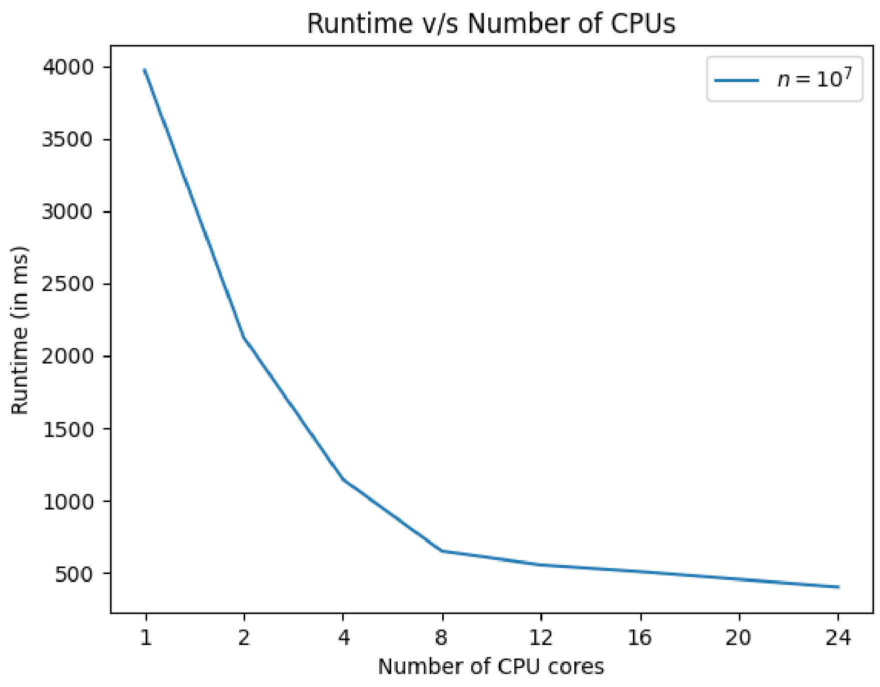

Figure 1.3: $n = 10^7$

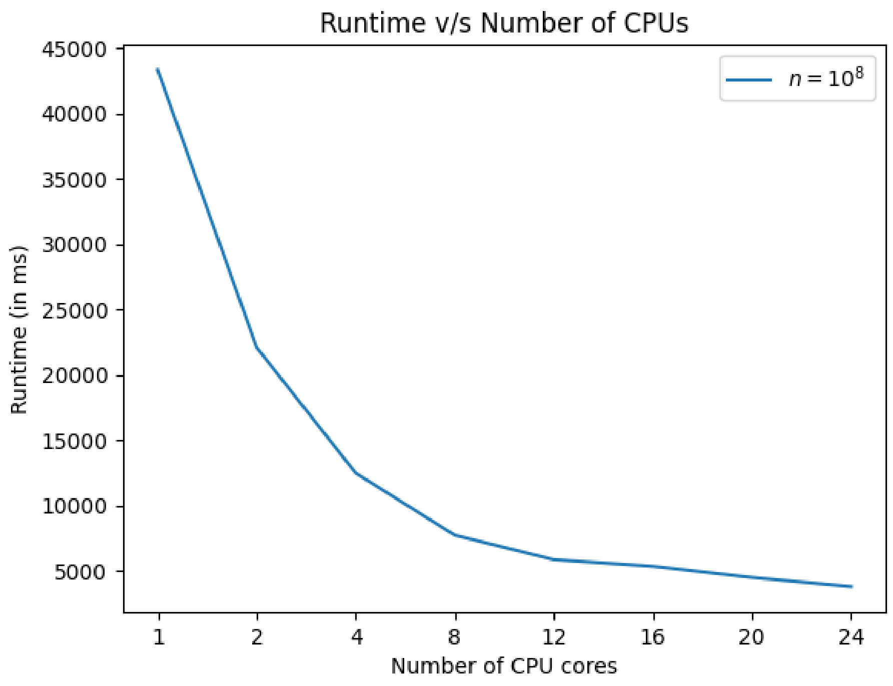

Figure 1.4: $n = 10^8$

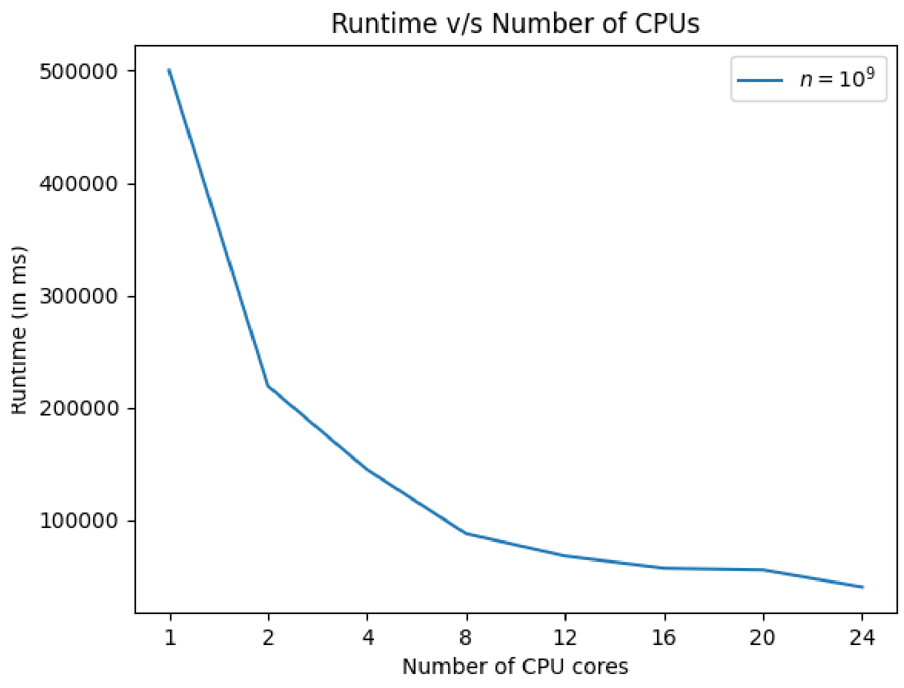

Figure 1.5: $n = 10^9$

- It can be seen from the figure that by fixing the number of cores, and varying the value of n leads to an almost *linear* rise in runtime. This is because the sequential sorting algorithm (*Quick Sort*) has $O(n \log n)$ time complexity on average (random data).
- Now, if I fix the size of array, and vary the number of CPU cores (and consequently number of threads), I observe that the runtime continually decreases. This is evident from Figure 1.1 to 1.5. The maximum speedup achieved is **12.29** for $n = 10^9$ and 24 cores. The maximum efficiency achieved is **1.14** for $n = 10^9$ and 2 cores. One common trend in all the plots is that for a given core count, speedup (and hence efficiency) is increasing with increasing value of n. For example, for core count = 24, the speedup values for Figure 1.1 to 1.5 are given in order: 7.46, 9.81, 9.84, 11.31, 12.29. Hence, the increase in speedup with increase in n can be seen for a given core count. This tells that as more data is provided, the program is able to better utilize the number of cores available to it. Another point that I observe here is that if I fix n and vary number of cores, I observe a decrease in efficiency of the program. For example, for $n = 10^9$, the efficiency for different number of CPUs is given in order: 1.14(2), 0.86(4), 0.71(8), 0.61(12), 0.54(16), 0.48(20), 0.51(24). The number in bracket indicates the number of cores used. Hence, it can be seen that my program is not strongly scalable, as the efficiency is going down with increasing number of cores. Combining this fact with the previous observation, I can conclude that my code is *weakly scalable*, as the core utilization increases with increasing size of input.