Concurrent Programming - assignment 1

Matej Majtan - s184457, Jun Chen - s202781

September 2020

1 Problem 1

The graph below (fig.1) shows changes of the running time while the number of runs increases. The running time for the first 2 runs are substantially bigger than the latter runs.

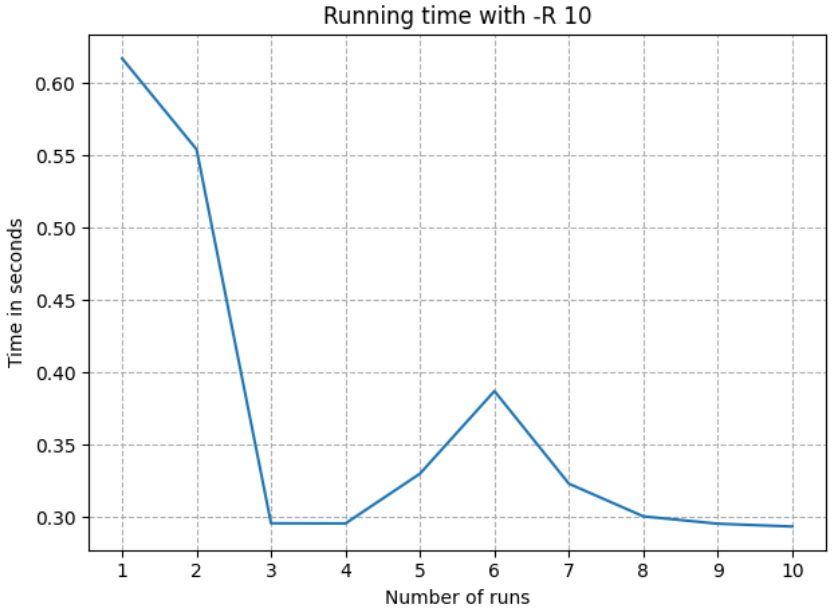

Figure 1: running time with -R 10 without warm up

This behaviour could be due to lazy class loading when the Java runtime environment only loads classes into memory the first time they are referenced in order to reduce memory usage. Another potential reason for the decrease in the running time could be because JVM uses Just-In-Time(JIT)

compilation to improve the performance of Java programs by compiling byte codes into machine code at runtime, which means that it optimizes the execution of repeatedly executed code.

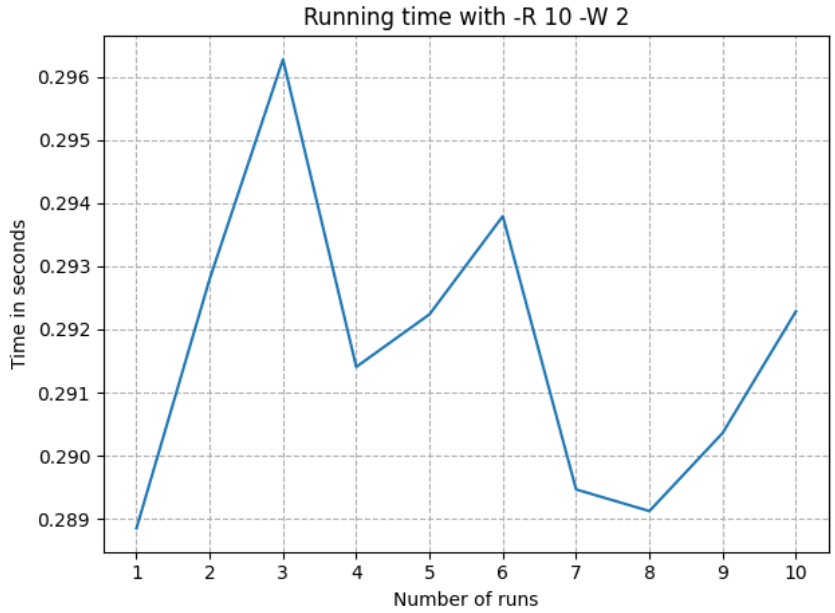

Figure 2: running time with -W 2 -R 10

In order to prove our assumptions, we set parameter `-W` as two to let JVM run two times before recording the running time. As a result, the fluctuation of running time shown in figure 2 is smaller than the one in Figure 1, and we can see a distinct improvement of running time for the first two runs since JVM has run the algorithm two times in advance so that latter runs can use the already optimized code to run faster.

2 Problem 2

In order to improve the running time the workload of the algorithm was split into several tasks. To ensure that the workload is evenly distributed, the input text is split into number of parts corresponding to the number of tasks. Each task searches extra characters on top of the allocated size in the end (the length of pattern minus one), and the next task starts searching ahead of where the previous part was supposed to end. This ensure that the system will check the overlap of two parts of the text processed by the tasks to avoid missing match or duplicated match.

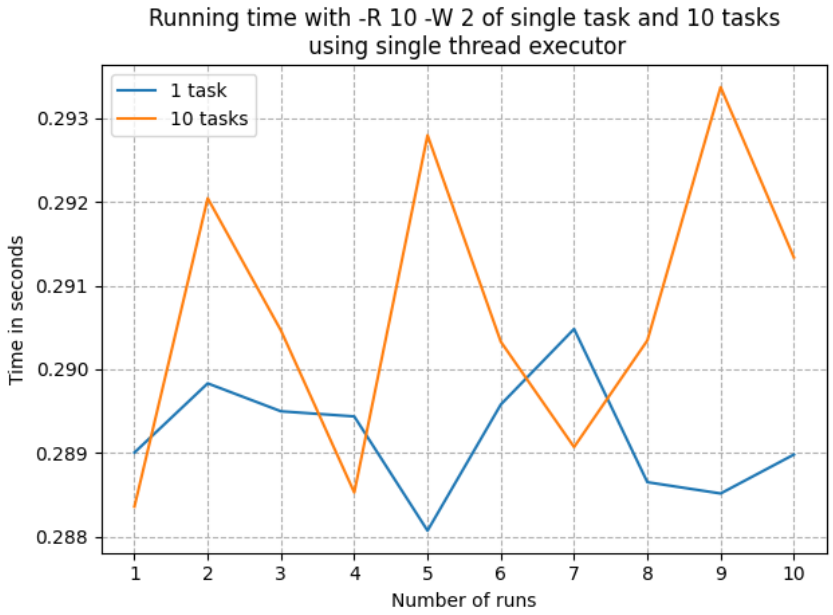

Figure 3: running time with -R 10 -W 2 of single task and 10 tasks using single thread executor

The figure 3 shows the measured times for algorithms that were both running using the Single Thread Executor, one running a single task (blue) and one running 10 tasks (orange). When running the multi-tasked code using a single thread executor, we have observed that the speed-up of the execution time from single-tasked to multi-tasked threads was actually around 1. Speed-up is the ratio of the running time using single task to the running time using multitasks, which means single task runs as fast as multitasks when using single thread executor.

This is because when only one thread is available, each task must wait for the current ongoing task finished before it can start, and that means though the total task is divided into multiple smaller tasks, using multitasks cannot outperform using single task. Besides every two tasks of the multitask algorithm have overlapped characters to check, that makes the total work load more than the one of single task algorithm.

3 Problem 3

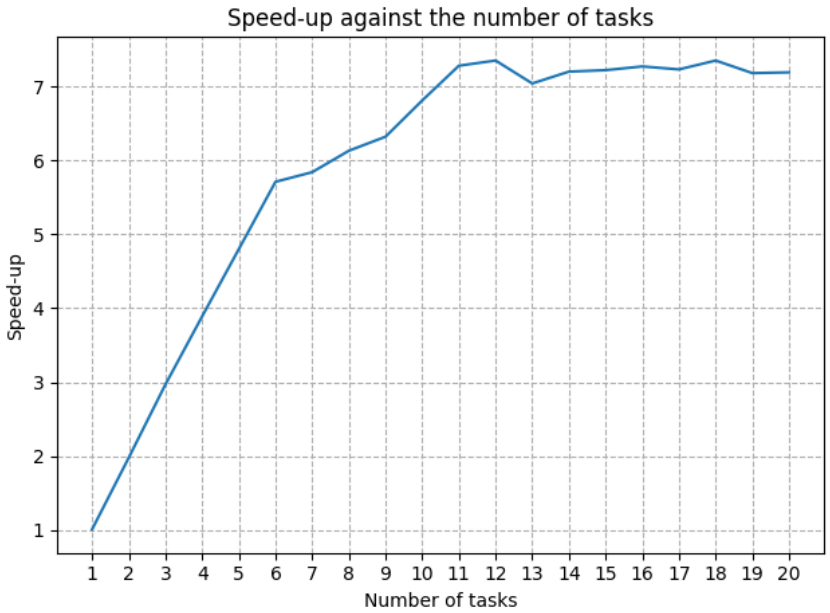

Figure 4: Speed-up against the number of tasks

As shown in Figure 4 the change of speedup seems linear from 1 task to 6 tasks, and the rate of change in speedup decreases from 6 tasks to 11 tasks. When the number of tasks exceeds 11, speedup fluctuates around 7.2. The most probable reason for this is the CPU itself. All the experiments were run on Intel i7-9750H CPU which has 6 cores with hyperthreading. This means that the CPU provides the system with 12 separate logical processors. This explains why the speed-up reaches maximum around the 12 tasks as more tasks would result in sequential execution of the tasks since only 12 tasks can be executed in parallel.

4 Problem 4

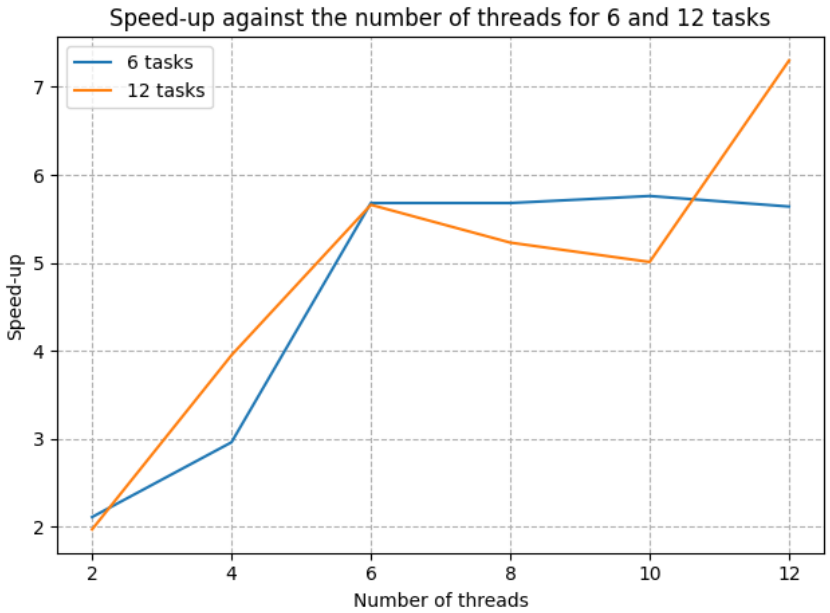

Figure 5: speedup against the number of threads

Figure 5 shows speed-up against the change of number of threads for 6 tasks and 12 tasks. For 6 tasks the speed-up reaches its peak at 6 threads, because all 6 tasks can run concurrently by using 6 threads and having more than 6 threads cannot improve the speedup since the final speedup is solely based on the running time of the 6 occupied threads.

However in the case of 12 tasks, we need to look at the CPU again. The CPU has 6 cores and 12 threads. This means that if we have 12 tasks and we allow up to 6 threads we get very nice performance for 2, 4 and 6 threads since all divide 12 evenly e.g. in case of 2 threads we assign 6 tasks to each thread, in case of 4 threads we assign 3 tasks to each thread, in case of 6 threads we assign 2 tasks for each thread. However when we get up to the 8 tasks we are effectively splitting the power of CPU to 8 threads which can run 8 tasks in parallel however 4 of them need to run the rest of tasks which keeps half of allocated threads inactive. It is a similar case for 10 threads. However when we get to the case of 12 threads it is apparent that since all tasks can be executed concurrently the resulting speed-up is much more significant than the other cases.