Danmarks Tekniske Universitet

DTU

Mandatory Assignment 1

Authors

Aryan Dhaka - s231897
Jitendra Landa - s231888
Mustafa El-Madani - s215225

September 18, 2023

Contents

1 Problem 1

1.1 Execution Time Variation

The execution time stabilizes after the two initial executions to about 2 seconds. The initial executions are seen as warm-up runs with a higher execution time of about 3.7 seconds.

1.2 Plot

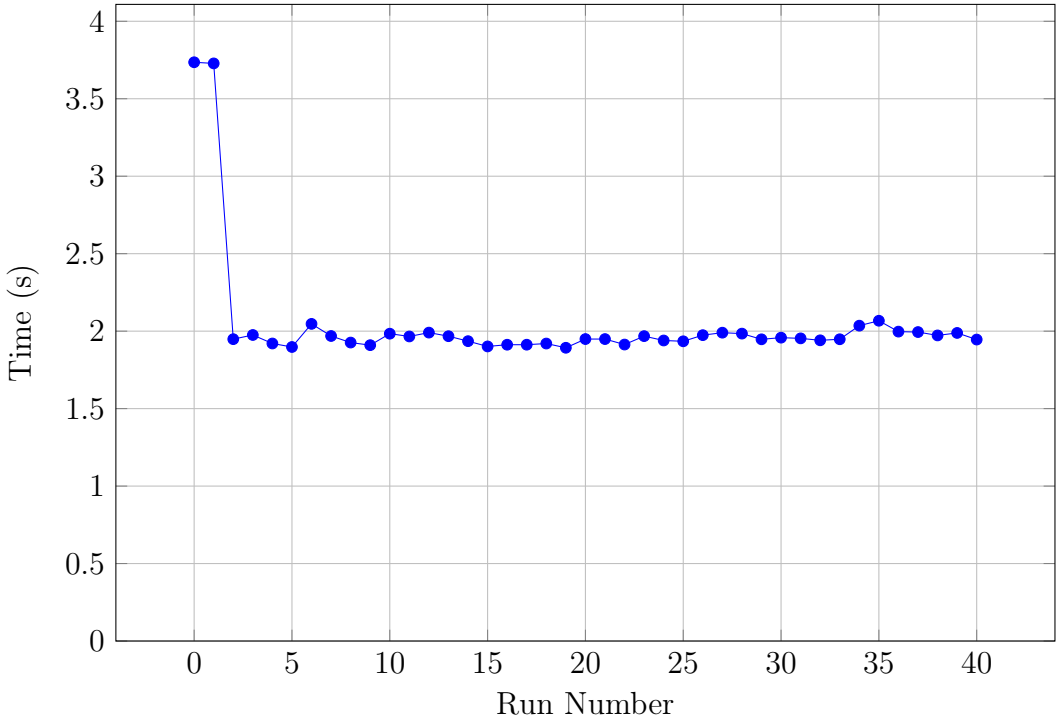


Figure 1: Time vs. Run Number

1.3 Warm-up Phenomenon

The warm-up phenomenon resulting in higher execution times initially observed here might be a result of multiple reasons. Some of them are;

1. **Runtime Execution:** The code isn't immediately translated into highly efficient machine code during startup. Instead, it begins in a less optimized form, like an interpretation or basic compilation. As the program continues to run, the system gradually identifies frequently used code segments and converts them into optimized machine code. This transformation process may require a few initial runs to reach its peak performance.

2. **Cache:** During the warm-up phase, the cache might be fully used up with data not corresponding to the executed file, leading to slower execution times.

3. **Resource Allocation by OS:** Operating systems allocate resources dynamically based on program demands. In the warm-up phase, the OS may be adjusting resource allocation, affecting execution times.

1.4 Problem Definition and Set Variables

The problem used for the testing is an analogous x file with a higher number of characters and the search pattern is a long string of "x"s separated by "|"s.The phrase and search text are long enough to have the desired execution time orders and the resultant searches are considerable as well.

The value of <W> is selected as 10, with two considerations from the plot: the first two, due to the higher execution time, and the rest as a precautionary measure which were observed with different tests and patterns.

Meanwhile, <R> is chosen as 15 based on a significant number of runs and the overall considerable in the total time taken.

2 Problem 2

2.1 Splitting of Tasks

The task splitting for concurrent execution is done as follows

- For division of work between the ntasks, we divide the text into n parts (each having approximately [total characters/n] characters) and then use the search algorithm on each of the divided subtexts.
- Mathematically,

$$Total\ length\ of\ text\ =\ l$$

$$Number\ of\ tasks\ =\ n$$

$$Length\ of\ Search\ pattern\ =\ t$$

$$Length\ of\ subtext\ (First\ n-1\ tasks)\ =\ [l/n]\ +\ t-1$$

$$Length\ of\ last\ subtext\ =\ l\ -\ (n-1)*[l/n]$$

  Where $[x]$ represents GINT of x

- Each subtext also takes some additional characters that overlap with the next part to account for cases where the pattern is split between adjacent subtasks
- The workload distribution in this program is relatively even as each task does the same pattern search on approximately the same number of characters.
- There is a small overlap dependent on the length of the pattern to be detected but since the overlap is the same for each task except the last, the workload remains the same.
- An upper limit exists for the number of tasks in which the text can be split that is given by [total characters/length of pattern] to have at least one search without overlap going beyond the adjacent subtext

2.2 Speedup and Limitations

- There is a notable absence of speedup in this part of the problem as the program is still executed and confined to one thread, hence splitting the work does no benefit as no concurrency is allowed and the subtasks can't be executed at the same time.
- Sequential execution achieves identical results as one big single execution which is observed through the plots as well. This is the limitation that is present in this part.
- Expanding the number of tasks to a substantial quantity fails to provide any insights as the graph is a mere straight line and merely consumes additional time, rendering it redundant.

2.3 Plots

The supposition is supported and verified by the plots

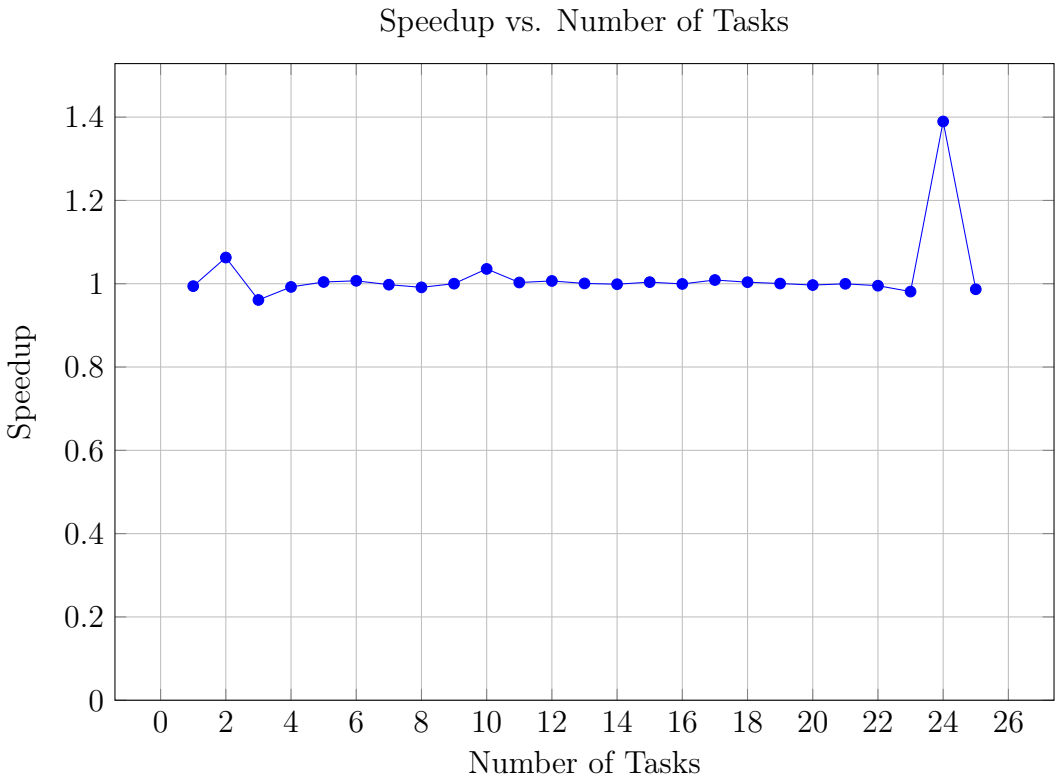


Figure 2: Speedup vs. Number of Tasks

3 Problem 3

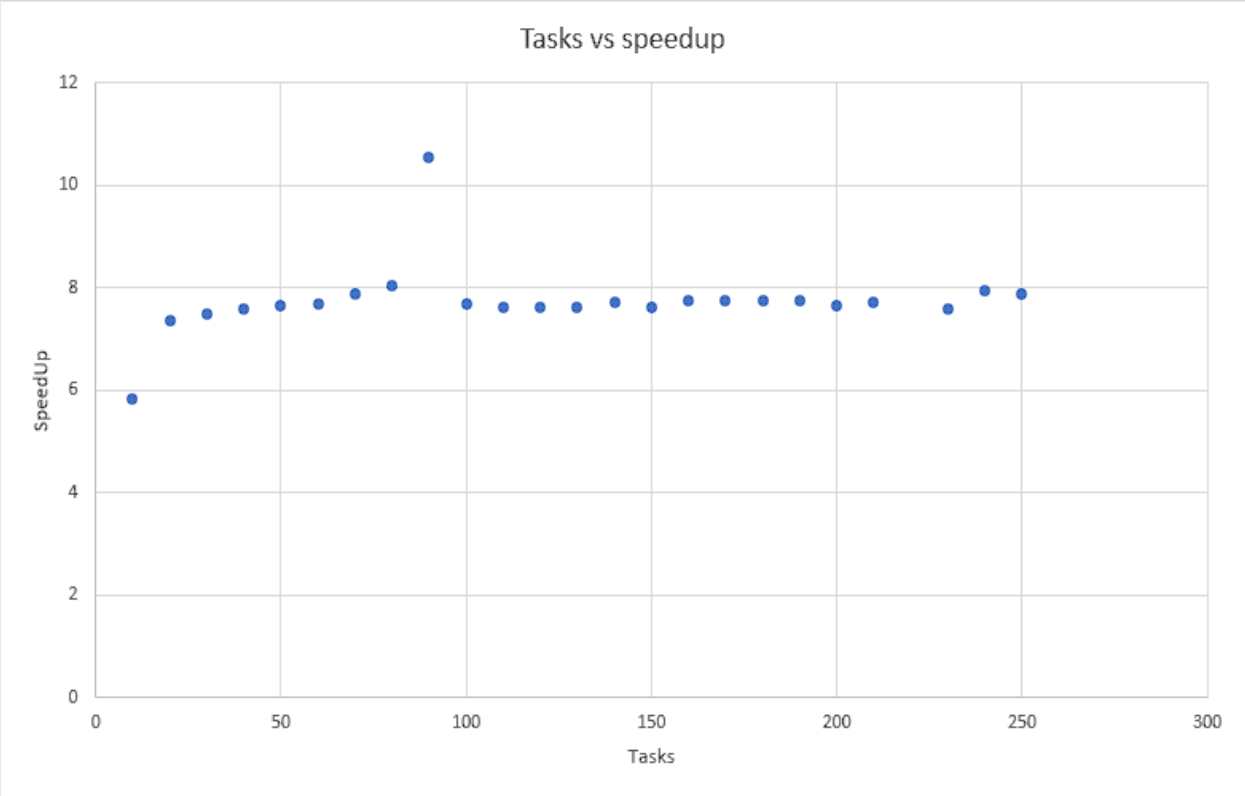
3.1 Experiment

An experiment is conducted using the -Ec option to execute each task on its own thread. This experiment is conducted on an AMD Ryzen 7 7800x3d CPU which has 8 cores, 16 threads and 4.2GHz clock speed. The goal is to see what does the increase in the number of tasks (threads) will affect the speedup. This experiment will use the the parameters defined in problem 1. There should be 9 occurrences in 9940009 characters.

3.2 Results

Figure 3 shows the result of the experiment where the number of tasks is on the x-axis with 250 data points, and the resulting average speedup on the y-axis.

Figure 3: The effect increasing the tasks on the speedup

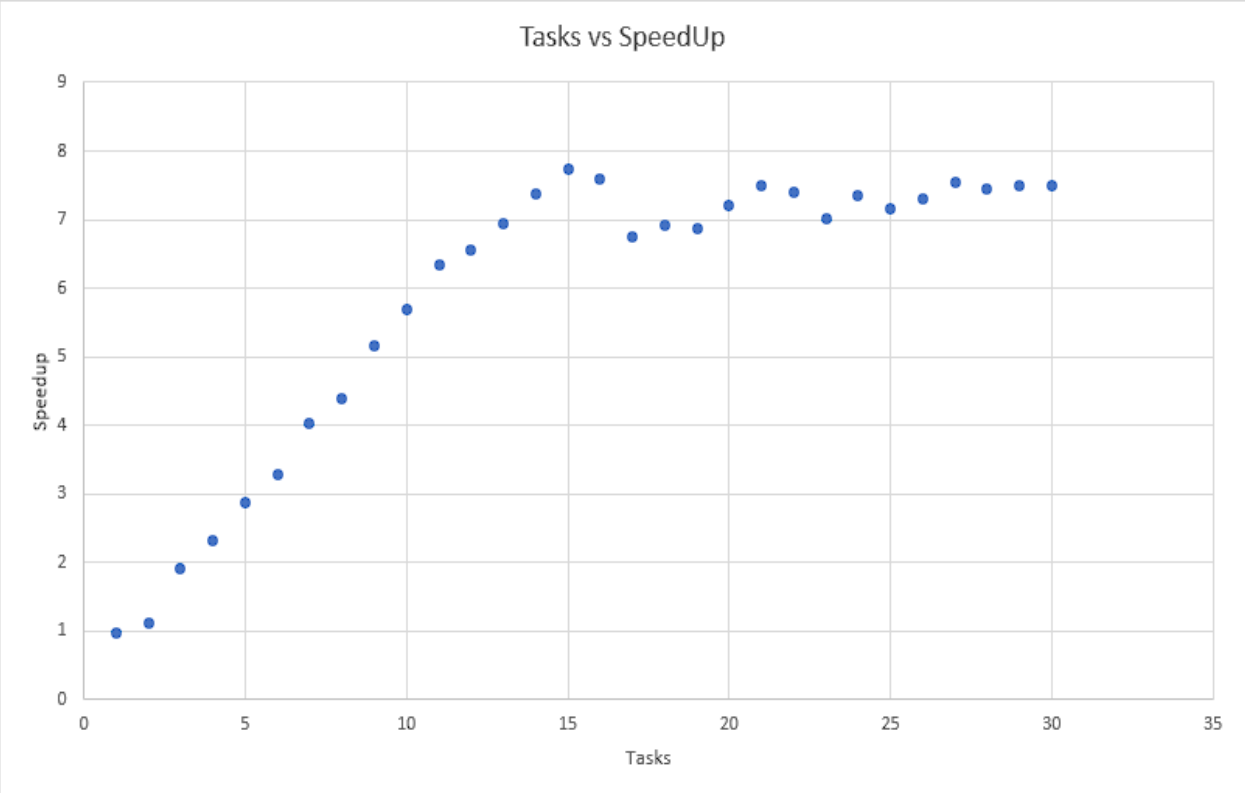


The speedup remains relatively constant as the tasks increase (the data point of 220 tasks is removed due to it being an outlier with the speedup value of 425). However, this only occurs after some number of tasks, if figure 3's range is limited to 30 (see figure 4), we can see that there is an increase in speedup. The speedup values shows the influence of parallel programming on the execution time, the larger the speedup value the faster the

tasks are executed concurrently compared to only using 1 task. This means that at first when there is an increase in the number of tasks (threads), the program executes faster up to a certain point, then the marginal speedup benefit of adding an extra task will decrease. This is expected since, after a certain point, adding more tasks may not lead to significant speedup due to the character length of the subtasks being too small for a genuine search. For example, searching a 10-character pattern in a 100-character text. When the number of characters per sub-task is less than 10, say 8, even the first search on the subtext includes adjacent subtexts and a lot of repetitive searches start to occur thus no or very little speedup is observed beyond this point.

Moreover, the speedup seems to converge on a value close to 8, meaning that single task is at least 8 times slower than multiple tasks.

Figure 4: The first 30 data points

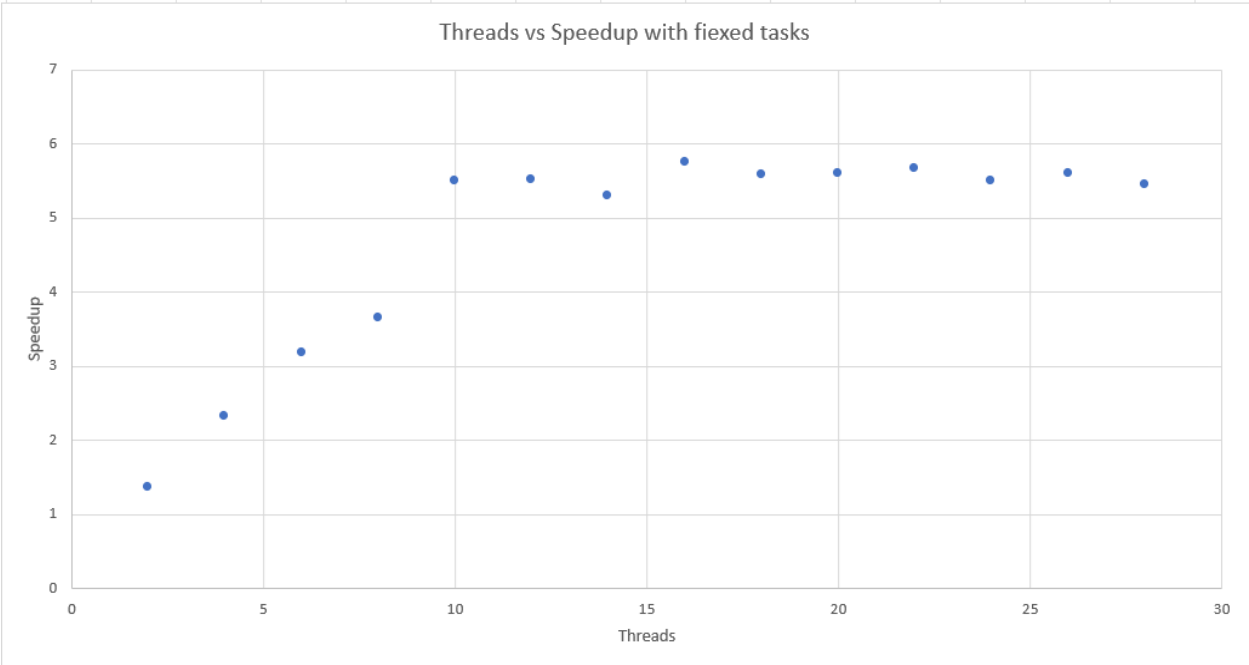


4 Problem 4

4.1 Effect of Threads

In the beginning, increasing the number of threads resulted in a decrease in the execution time until the number of tasks nears the number of tasks. Afterward, the increase of threads will do very little in terms of execution time of the program. This is due to the fact that there are not enough tasks to distribute between the threads, so some threads will be created and do nothing.

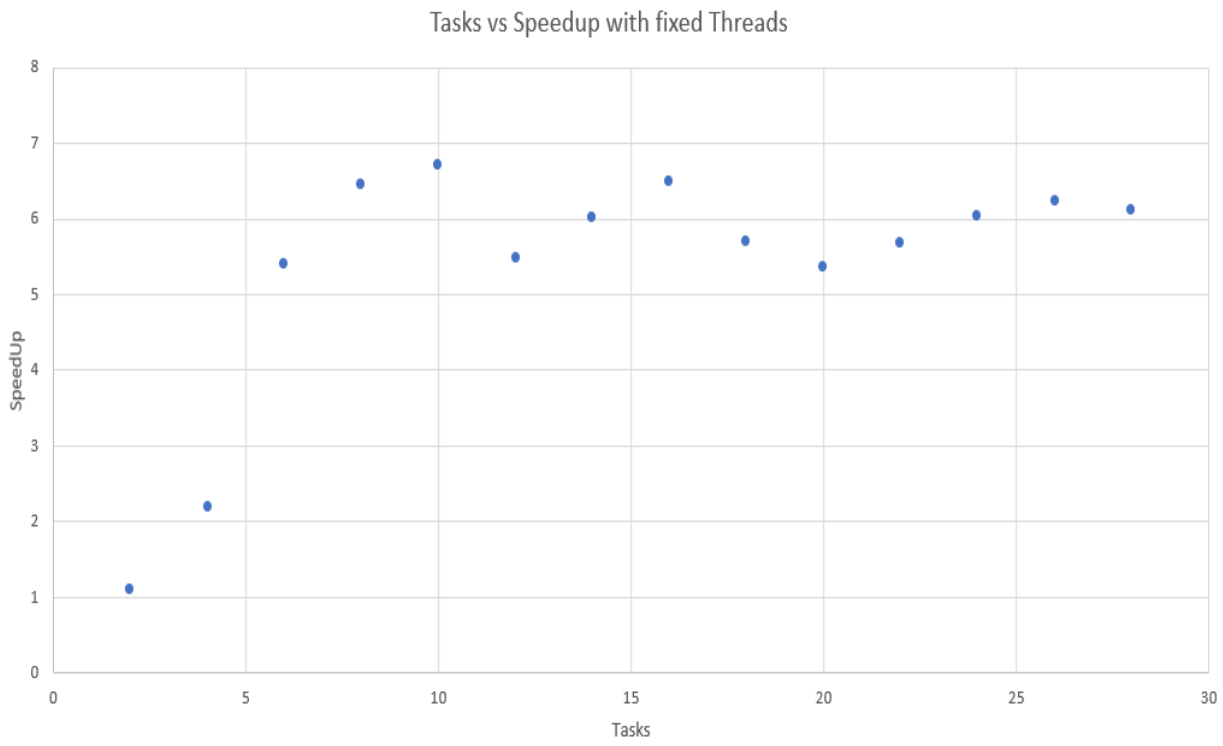
Figure 5: The effect of number of threads on execution time, with 10 tasks



4.2 Effect of Tasks

The same phenomenon can be observed with the tasks as well for a constant amount of threads i.e when the number of tasks is less than the number of threads, there is an increase in the the speedup until it nears the fixed amount of threads which is 10, then the speedup slows down. This is expected as there are more tasks than threads so each threads will have a queue of tasks that must be done. Moreover, as discussed in problem 3 some tasks are meaningless, they will just slow down the execution, and this might cause a slight decrease in speedup. Furthermore, one can expect that increasing the number of tasks by a single task more than a multiple of the number of threads will decrease the execution time, for example, one can expect that with 10 threads, 11 tasks and 20 tasks will take the same amount of time(assuming all tasks are equal).

Figure 6: The effect of number of tasks on execution time, with 10 threads

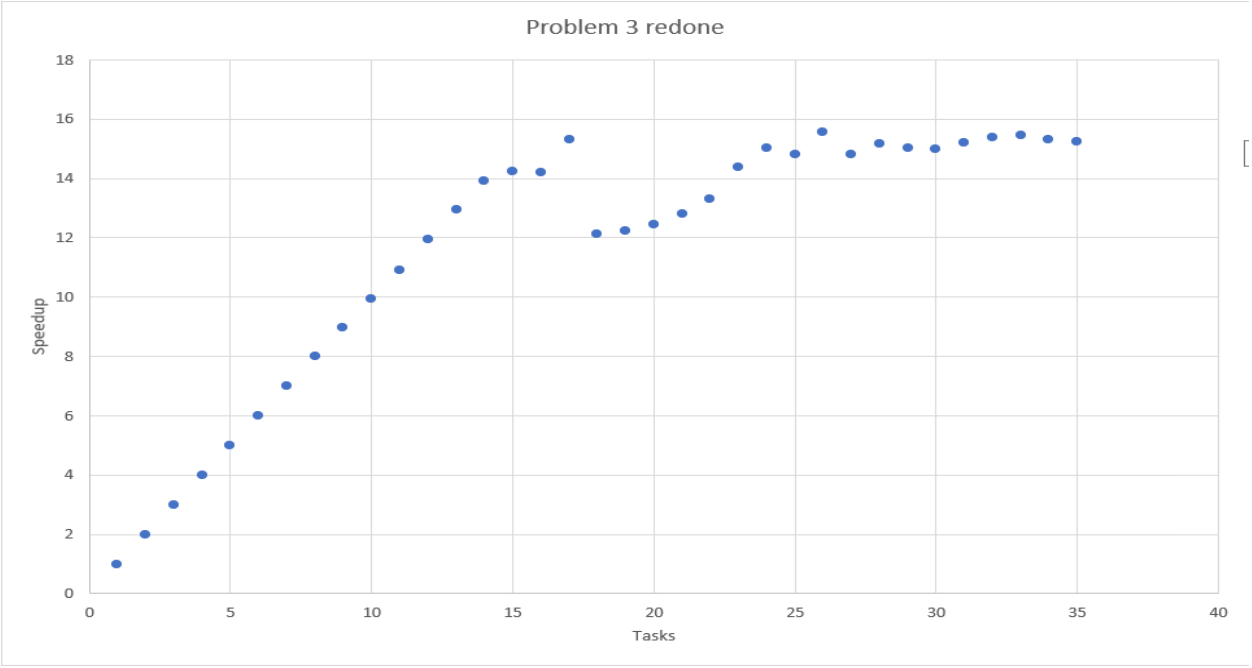


5 Problem 5

5.1 Problem 3 on HPC machine

We can clearly see that the shape of the graph obtained i.e figure 7 is same as the shape of the graph obtained in problem 3 i.e figure 4. Although, The graph shape is similar, we can see the difference in magnitude i.e the highest attained speedup using HPC machine having 20 cores(found using lscpu) is almost double compared to the one attained using home computer having 8 cores. Moreover, We can observe that after the peak in speedup, the speedup is oscillating, but steadily increasing, this may be due to the fact that as tasks increase more tasks that prone to are created thus decreasing the marginal benefit of adding more tasks(threads).

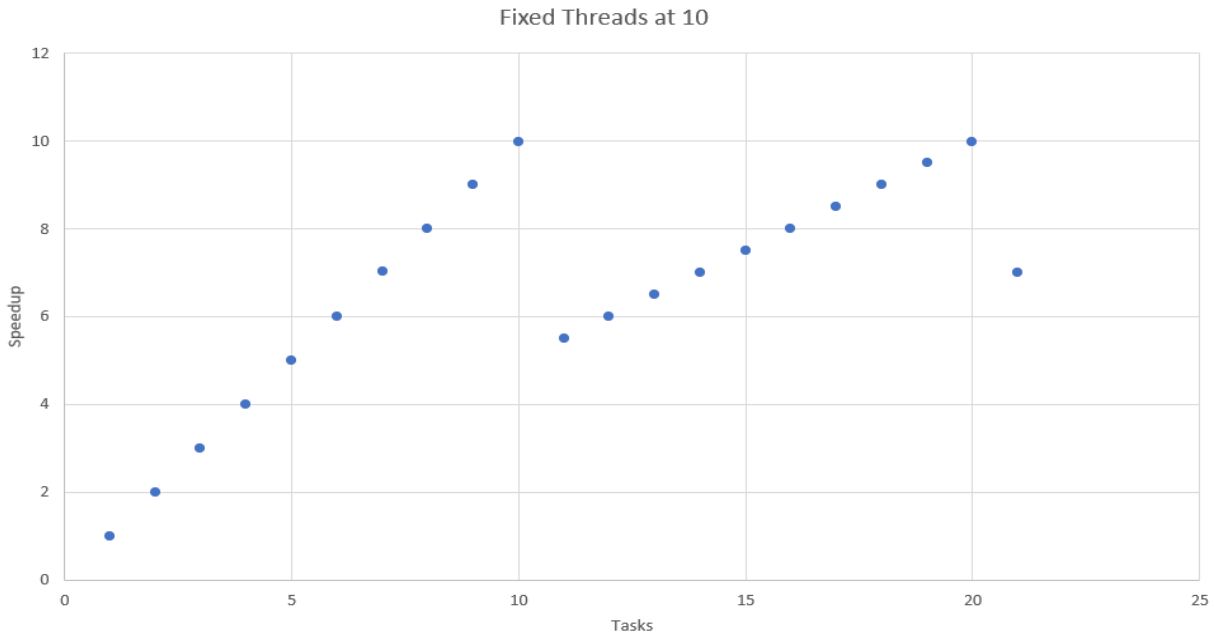
Figure 7: Result of problem 3 in HPC machine



5.2 Problem 4 on HPC machine

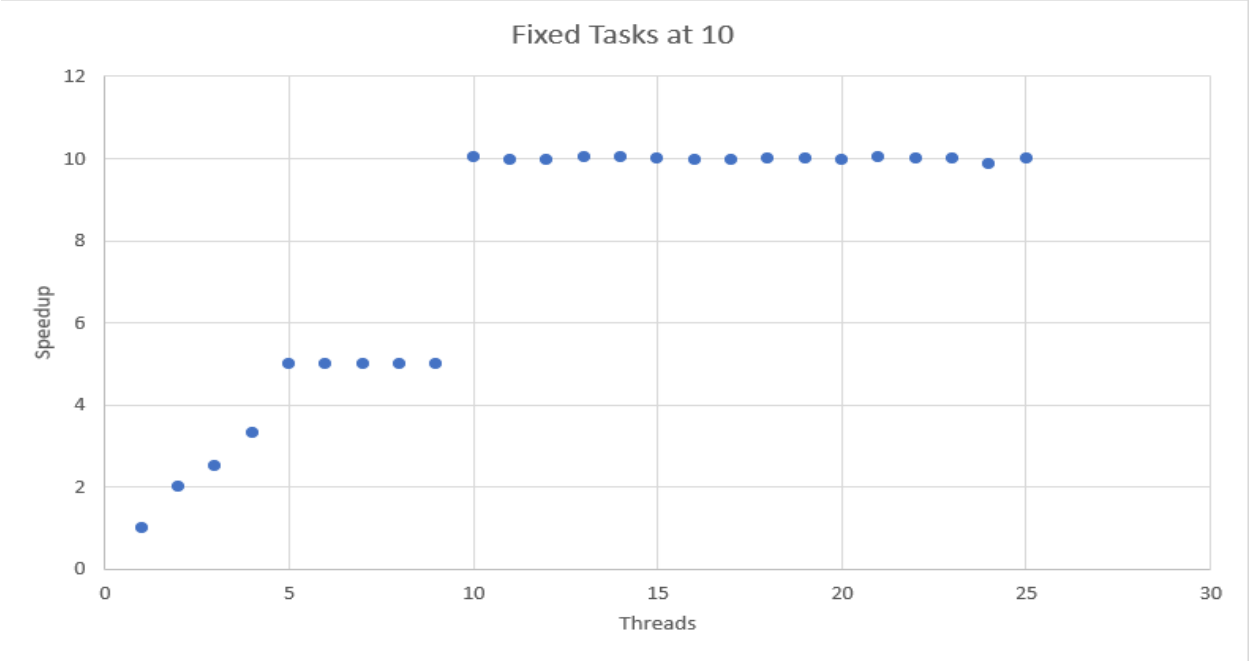
In the figure 8 we can see an increase in speedup as the number of tasks increase, this is expected as more work being divided on the threads so they can work more efficiently and concurrently. Moreover, for every 10 tasks added we can see a decrease in speedup, this is expected as explained in problem 4. For example if we have 11 tasks, then 10 tasks will execute concurrently with the 10 threads and the one left over will execute on its own, hence the decrease in speedup since at least 1 thread will have to do 2 cycles of work.

Figure 8: Result of problem 4 in HPC machine with fixed threads



The following figure 9 shows the effect of increasing the number of threads with the number of tasks being constant. At the start, from 1 to 5 threads, there is a linear increase in the speedup this is due to the fact that for every new thread added, the maximum number of tasks each thread must do decreases, for example at 2 threads each thread must do 5 tasks, but on 3 threads the maximum number of tasks a thread will do is 4, until 5 where adding a thread will decrease the maximum number of tasks a thread must do. However, this changes at 10 threads, since the number of threads and tasks are equal there is a large jump in speedup, this is because the maximum number of tasks each thread will do is 1. Moreover, an increase of the number of threads beyond 10 will not result in a meaningful increase in speedup, this is due to the fact that creating more threads beyond number of tasks present will result in threads that do nothing as discussed in problem 4.

Figure 9: Result of problem 4 in HPC machine with fixed tasks



6 Conclusion

In this assignment, we implemented and looked at concurrent execution and its implications in Java. The thread pool framework was used and closely analyzed how a basic search program behaved under different experiments and situations. Through this exploration, we gained insights into the challenges and benefits of concurrent processing.

Reflections on Using Concurrency for Improving Performance:

- **Execution Time Change:** Observed that execution time for the same program on the same machine can vary due to OS resource allocation, Empty cache, etc. This requires careful measurement and analysis to prevent wrong conclusions
- **Warm-Up Phenomenon:** Initial runs exhibit longer execution times compared to subsequent ones and the need to exclude these runs from our analysis.
- **Speedup:** The necessity of splitting tasks on different threads to actually improve performance and realize concurrent programming is affected through problems 2 and 3. Simply splitting the task and retaining the same process has no observable changes which is observed with the speedup being 1.
- **Concurrency Challenges:** Experimentation with different thread pool configurations, including a cached thread pool and a fixed thread pool. While cached thread pools allowed flexibility, fixed thread pools provided more control but required careful tuning.
- **Performance on HPC Machines :** Explorations on HPC machines, which offer a vast number of cores. Multi-user environments offer varied results due to resource sharing but having more threads improves performance.

In conclusion, concurrency is a powerful tool for improving program performance, especially when dealing with computationally intensive tasks that can be split without dependencies. However, it introduces complexities such as load balancing, thread management, etc. The choice of the right thread pool and thread management strategy plays a crucial role in harnessing the potential of concurrency. The assignment provides a real-life insight into basic concurrency and its wide application.