Assignment-1 Report

Group id - 40
Bhumanyu Goyal(B18012)
Vivek Kumar(B18092)
Harish Jaglan(B18115)

Problem2

1) **Deadlock Prevention**

   Deadlock has 4 requirements for existence:-

   - Mutual Exclusion, Hold & wait, No preemption, Circular wait

   If any of them is violated deadlock will never occur. The code violates the "*Hold and Wait*" condition, a student can get 0 or 2 spoons (1 spoon not allowed). Thus no student waits for a spoon while holding the other spoon.

   Code uses "*spoonBoth*" lock to allow only one student at a time to acquire locks on spoons. If a student can acquire both spoons, it locks both but if the student can't get both it waits until it can get both spoons using "*pthread_cond_wait*".
   So there is **no deadlock**.

   Some extra points/inferences:-

   - As students can have only 0/2 spoons, a student can get a spoon when both its neighbours have 0 spoons
   - "p*thread_cond_wait*" is signalled when a student releases its both spoons, a student signals both of its neighbours.

2) **Mutually Exclusive**

   A particular spoon can be used by only one student at a time. This is ensured in the code by allowing a student to pick a spoon only when it's not used by anyone.

   Code uses "*valid*" function to check neighbours.
   So there is **mutual exclusion**

3) **Starvation**

   Here starvation means that a student is waiting for a much longer time than other students to get spoons.
   To prevent starvation code uses following strategy:-

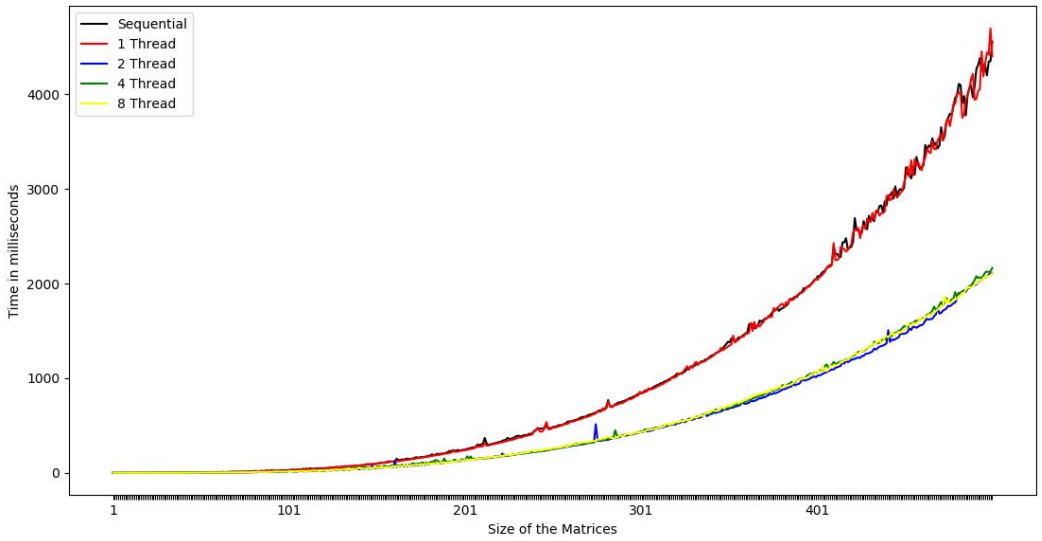
- Each student has some priority which is last timestamp at which student has used both spoons (using "*time*" function)
- Lower timestamp means waiting from longer time, thus more priority to get spoons
- When a student is called to acquire spoons, it first checks if its neighbours have been waiting for too long or not.
- If any of the 2 neighbours have been waiting since long time then don't take spoons and go to wait state through "p*thread_cond_wait*"
- "*valid*" function in code checks if neighbours have been waiting more than some threshold wrt this student.

Idea of correctness of preventing starvation:-

- Consider some continuous neighbours as starving and rest as getting much higher calls. So the students on the boundary of two regions and in the high call region would add into starving neighbours if not called (If they are called they would cond_wait on starving neighbours and remove starvation). If the starving region would keep on expanding until 1 or 2 students are left in the high call region, these 1 or 2 would then wait as per instructions for their neighbours and prevent starvation.
- Important aspect is to set the time difference between neighbours to give more priority to longer waiting neighbours which can be set as per requirements.

Problem3:

Below is the graph of the time taken by the sequential and multithreaded program for various sizes of the input matrices. We ran the program for some specific no of threads.



Discussion:

- In the algorithm of matrix multiplication, the elements of the answer matrix are calculated at different memory locations thereby allowing the flexibility to use different threads for calculating the elements.

- The distribution of the computational work among the threads is a design problem that can be tackled in different ways. One naive but efficient approach is to create a separate thread for each row(or column) of the matrix. Since there are only read operations on the matrices to be multiplied, race condition is avoided.

- Increasing the number of threads after a certain point doesn't improve the parallelism further. One possible explanation is some cores are already busy running the OS and other critical processes of the computer.

- Another observation that we made is using too many threads is not a feasible option as the time taken for context switching which was initially negligible turns out to the bottleneck and slows down the program. Moreover, there is only a certain number of threads the OS can create for the process.