Report - Lab 1

Part 2: Dining Students

Introduction
We were required to write a program to solve the Dining Students Problem by using locks. We then ran the code 5 times and every time Simulating 30 minutes. We then recorded the order of states change in different log files. Run the following command

```
g++ Q2.cpp -pthread && ./a.out
```

Parameter :
Simulation Time - 1800 seconds
Number of Students - 5
Number of Times the code was run - 5
Eating time for Students - 20 seconds
Thinking time for students - randomly between 1 to 2 seconds.

Statistics:
While running the code we kept a record of ll state changes and also noted the amount of time each student spends in a state. These Statistics helped us in analysing deadlocks, starvation etc.

Table 1: Eating Time for Each Student in every Iteration

Student	**Eating 1**	**Eating 2**	**Eating 3**	**Eating 4**	**Eating 5**	**Average**
1	**680**	**720**	**700**	**720**	**700**	**704**
2	**700**	**700**	**700**	**680**	**680**	**691**
3	**720**	**700**	**680**	**680**	**720**	**700**
4	**700**	**700**	**660**	**680**	**700**	**688**
5	**680**	**700**	**680**	**680**	**700**	**688**

Table 2: Waiting Time for Each Student in every Iteration

Student	Waiting 1	Waiting 2	Waiting 3	Waiting 4	Waiting 5	Average
1	**1.99605**	**1.94374**	**1.97545**	**1.78684**	**2.01863**	**1.944142**
2	**1.92040**	**1.90760**	**1.90339**	**1.93189**	**1.92829**	**1.918314**
3	**1.90507**	**1.92400**	**1.98575**	**1.98336**	**1.82937**	**1.925510**
4	**1.91143**	**1.93456**	**1.90530**	**2.04833**	**1.97381**	**1.954686**
5	**1.91942**	**1.99503**	**1.83810**	**1.88619**	**1.96658**	**1.921064**

Observations:
- The code got executed to completion all the five times which shows that there were no situations of deadlock. All the state changes could be successfully stored in the output file.
- All the students could eat for a significant number of times and there were not any big differences in the eating frequencies for them in all the five executions.
- The average waiting time was also less for all the students in all the five executions which can help us show that there were no instances of starvation for any of the students.

Conclusion:
Using the concepts of locks we are able to solve this problem. To eat, each student needs two spoons. So, when a student wants to eat, he will take one left spoon and one right spoon and then lock them and then start eating. But this approach can lead to deadlock. Imagine if each student acquires a lock on the left spoon then there will be no spoon left on the table but all the students are waiting for their right spoon. But notice that the above mentioned senario is technically the only way a deadlock can happen here and this happening has a very small chance. So even if we do not do anything for the deadlock specifically there is only a very small chance of that happening. But to prevent this what we did is that we simply added a timeout after which a lock on the spoon will be released and the student will go back to thinking or go on to Eating. Also, there was no case of starvation in our code because every student could eat for significant number of times. The threads in my program are not mutually exclusive as a spoon is a shared resources between 2 threads. My program does not follow any particular order / priority for allocation i.e. it randomly allocates spoons to students(that are requesting access) and thus due to symmetrical reasons the Average Eating time of all the students are nearly comparable.

Part 3

INTRODUCTION AND PROCEDURE-

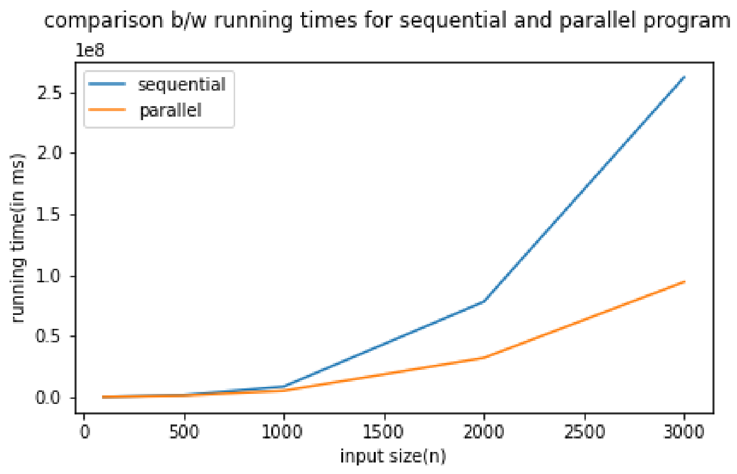
The codes provided along with the report accomplish the same task with two different approaches- one runs in a sequential manner whereas the other runs with the help of threads. The threaded program multiplies the matrices by dividing the task into 4 parts (using 4 parts is recommended for maximum efficiency according to the configuration of most processors). 4 threads run parallely and compute the 4 parts. Both the codes, upon completion provide the time elapsed during running. I have run the two codes 5 times for 5 observations ranging from n=1 to 3000. For each observation the average time is recorded and is plotted against the input size.

OBSERVATIONS-

Input size (n)	Time for sequential (ms)	Time for parallel(ms)
100	8431	7528
	8915	10959
	9103	8106
	16838	6214
	11883	9197
	Avg time: 11034.0	Avg time: 8400.8
500	1470346	1083354
	1489857	1206488
	1438404	1219718
	1471358	1152806
	1978831	1009665
	Avg time: 1569759.2	Avg time: 1134406.2,
1000	8422373	5077819
	8532437	5017101
	8420423	4925438
	8498529	5418184
	8455171	4920299
	Avg time: 8465786.6	Avg time: 5071768.2

2000	78370507	31387001
	77458849	31741606
	77781988	31924745
	78214283	32244623
	78163361	32739916
	Avg time: 77997797.6	Avg time: 32007578.2
3000	264262877	93337801
	261644892	97772982
	261994880	93286262
	259661238	93334772
	261886294	93407173
	Avg time: 261890036.2	Avg time: 94227798.0

GRAPH



INFERENCES AND CONCLUSION-

1) The sequential algorithm takes more time as input size increases as compared to the one using parallel processing.

2) The difference between times keeps on increasing as input size increases.

2) Using too many threads can cause memory overflow and can lead to increase in processing time as the CPU resources will not be sufficient to handle them.