High Performance Computing

(6CS005)

Portfolio

Student Id : 2039211

Student Name : Samir Husen

Module Leader : Mr. Jnaneshwar Bohara

Submitted on : Dec 26th, 2020

6CS005 Learning Journal - Semester 1 2020/21

Samir Husen 2039211

Table of Contents

1 Parallel and Distributed Systems

1.1 Answer of First Question

Thread is basic unit of CPU (Central Processing Unit) execution which have parallel execution paths contained by a process. Thread are mostly used when the tasks are essentially part of the same job and has no data and segment heap. It also improves the performance and allows sharing resources.

Thread are designed to solve the problem of processor utilization and resource starvation.

1.2 Answer of Second Question

The process scheduling policies are as listed below:

- Time slicing: In time slicing a task executes for the predefined slice time to occupant the pool of ready task and later the scheduler determines which task to execute first based on significance and other considerations.

- Pre-emptive scheduling: In pre-emptied scheduling the tasks are recognized with their priorities and is not executed until any high priority task enters.

Pre-emptive scheduling is preferred over time-slicing because it avoids the chances of a process controlling CPU time by pre-emptively switching to a higher priority task if one enters.

The behaviour of Java threads is completely dependent on the underlying scheduling scheme. The scheduling competence changes corresponding to the policy all other threading behaviours are program and policy independent.

1.3 Answer of Third Question

S.N	Centralized System	Distributed System
1.	Centralized system has only single element with non-independent components with single point failure.	Distributed system has several elements with multiple point failure.
2.	All resources in this system are available.	Some resource in this system may not be available.
3.	The components in this system are shared by users all the time.	The components in this system are not shared by users.
4.	The software in this system runs in a particular process with single point control.	The software runs in simultaneous processes on multiple processors with multiple points of control.
5.	In the centralized system if the main system fails, everything stops	In distributed system no harm is caused to other units if the system fails

1.4 Answer of Fourth Question

Transparency in distributed system is a characteristic that hides from the users that the processes and resources are physically distributed across multiple computers. It hides different form of such as access, location, replication, failure, migration and more. Transparencies have a related cost, and it is exceptionally important for the distributed system users to be aware of this fact.

1.5 Answer of Fifth Question

Process 1: B=A+C

Process 2: B=C+D

Process 3: C=B+D

Process 3 is flow-dependent on process 2 as the output of process 2 is needed by process 3

Process 3 is anti-dependent on process 1 as process 3 improves C which is needed by process 1

Process 2 is output reliant on process 1 as process 1 and 2 improves B

Since both anti-dependency and output dependency is named dependencies, they can be resolved by using temporary variables.

- **Source code**

```
int main (int argc, char** argv)
{
	int A = 1;
	int B = 2;
	int C = 1;
	int D = 4;

	int B1 = A + C;
	int C1 = C;

	B = C1 + D;
	C = B1 + D;
}
```

1.6 Answer of Sixth Question

The programs given in table produces the same 500000 as output.

The first program runs five threads and wait for them to finish.

Where the second program creates a thread and waits for the process to compete and repeats the same process five times.

Subsequently each program implements the same thread function which increases the same global counter, and the result stays the same while giving output.

2 Applications of Matrix Multiplication and Password Cracking using HPC-based CPU system

2.1 Single Thread Matrix Multiplication

- The complexity of the algorithm is $\boldsymbol{O(n^3)}$ which multiplies two matrices A and B and store the outcome in C.

- The three different methods to speed up the matrix multiplication algorithm are as listed below:

1. Divide and Conquer method

In this method the time complexity for the algorithm stays the same and this method is also simple to obtain the result. Suppose we have two square matrices M and N. Then we divide the matrices into four submatrix and calculates the result in matrix C. The calculation can be done as,

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad N = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$P = \begin{bmatrix} A & B \\ C & D \end{bmatrix} * \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$P = \begin{bmatrix} A*E+B*G & A*F+B*H \\ C*E+D*G & C*F+D*H \end{bmatrix}$$

P is the result matrix from above calculation which involves 8 multiplication and 4 addition. The time complexity of the above calculation can be written as,

$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$

2. Strassen's Matrix Multiplication Method

Strassen's matrix multiplication is like divide and conquer method, but it reduces the number of multiplication steps during the matrix calculation using the formula developed by Strassen's. The formula reduces the recursive calls up to 7 which is lesser than the divide conquer method. Suppose we have two square matrices M and N, and O is the result matrix.

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad N = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$P = \begin{bmatrix} A & B \\ C & D \end{bmatrix} * \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$Here, \qquad q1 = A(F - H) \qquad q2 = (A + B)H$$

$$q3 = (C + D)E \qquad q4 = D(G + E)$$

$$q5 = (A + B)(E + H)$$

$$q6 = (B - D)(G + H)$$

$$q7 = (A - C)(E + F)$$

$$Then, \qquad P = \begin{bmatrix} q5 + q4 - q2 + q6 & q1 + q2 \\ q3 + q4 & q1 + q5 - q3 - q7 \end{bmatrix}$$

The time complexity of the above calculation can be written as,

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

3. Permutation of Naïve and Divide and Conquer method

The permutation of Naïve and Divide and Conquer methods can stop divide and conquer at a particular intensity which enhances the whole speed of the matrix multiplication calculation.

- **Pseudo-code for the improved algorithm**

```
BEGIN

function MULTIPLICATION(A, B, C, M, N, P):
       if M == pow(2, DEPTH) AND N == pow(2, DEPTH) AND P == pow(2, DEPTH):
             loop i=1:N:
                    loop j=1:M:
                           C[ i ] [ j ] += A[ i ][ j ] * B[ i ][ j ]

       else
              A11 = A[0:N/2][0:N/2]
              A12 = A[0:N/2][N/2:N]
              A21 = A[N/2:N][0:N/2]
              A22 = A[N/2:N][N/2:N]

              B11 = B[0:N/2][0:N/2]
              B12 = B[0:N/2][N/2:N]
              B21 = B[N/2:N][0:N/2]
              B22 = B[N/2:N][N/2:N]

MULTIPLICATION (A11, B11, C11_1, N/2, P/2, M/2);
MULTIPLICATION (A11, B12, C12_1, N/2, P/2, M/2);
MULTIPLICATION (A21, B11, C21_1, N/2, P/2, M/2);
MULTIPLICATION (A21, B12, C22_1, N/2, P/2, M/2);
MULTIPLICATION (A12, B21, C11_2, N/2, P/2, M/2);
MULTIPLICATION (A12, B22, C12_2, N/2, P/2, M/2);
MULTIPLICATION (A22, B21, C21_2, N/2, P/2, M/2);
MULTIPLICATION (A22, B22, C22_2, N/2, P/2, M/2);

C[0:N/2][0:N/2] = C11_1 + C11_2
C[0:N/2][N/2:N] = C12_1 + C12_2
C[N/2:N][0:N/2] = C21_1 + C21_2
C[N/2:N][N/2:N] = C22_2 + C22_2

END
```

- The code is included under the folder Task 2_1

- **Time performance of improved single thread matrix multiplication based on Divide and Conquer Method**

REPETITIONS	SECONDS	NANOSECONDS
1	4.380s	4380000000ns
2	4.333s	4333000000ns
3	4.162s	4162000000ns
4	4.198s	4198000000ns
5	4.229s	4229000000ns
6	4.159s	4159000000ns
7	4.134s	4134000000ns
8	4.152s	4152000000ns
9	4.261s	4261000000ns
10	4.416s	4461000000ns
MEAN TIME	**4.242s +/- 0.097s**	4242000000ns

- **Hypothesis and analysis**

Both matrix multiplication methods are same but there is only difference of time because Divide and Conquer method has the lesser number of multiplication steps.

The size of matrices indices is supposed as the power of two for the easy multiplication.

Loop control and loop test instruction counts are highly reduced for divide and conquer method improving computation time

Assuming an A x A square matrix,

Total of similarities and incrementations in Naive method

$$2n \times 2n \times 2n = 8n^3$$

Total of similarities and incrementations in Divide and Conquer method till the depth of 'k'

$$equals\ to\ 2^k \times \left(\frac{2n}{2^k} \times \frac{2n}{2^k} \times \frac{2n}{2^k}\right)$$

$$= \frac{8n^3}{2^{2k}}$$

$$= \frac{8n^3}{4^k}$$

Meanwhile the number of similarities comes reduces approximately 4^k the algorithm implementation time is reduced equally with the lesser memory consumption. Both divide and conquer and naive algorithms, the number of multiplications and additions are the same, the time gain is higher in improved version is because the reduction in the number of repetitions.

2.2 Multithreaded Matrix Multiplication

- The code is included under the folder Task 2_2

- **Time performance of multithreaded matrix multiplication**

REPETITIONS	SECONDS	NANOSECONDS
1	1.302s	1302000000ns
2	1.648s	1648000000ns
3	1.475s	1475000000ns
4	1.516s	1516000000ns
5	1.466s	1466000000ns
6	1.502s	1502000000ns
7	1.461s	1461000000ns
8	1.399s	1399000000ns
9	1.415s	1415000000ns
10	1.542s	1542000000ns
MEAN TIME	**1.473s +/- 0.087s**	1473000000ns

- **The table of comparison between the original and the multi-threaded matrix version with mean time**

REPETITIONS	SINGLE THREADED MATRIX MULTIPLICATION		MULTI-THREADED MATRIX MULTIPLICATION	
	SECONDS	NANOSECONDS	SECONDS	NANOSECONDS
1	8.835s	8835000000ns	1.302s	1302000000ns
2	7.983s	7983000000ns	1.648s	1648000000ns
3	6.854s	6854000000ns	1.475s	1475000000ns
4	6.820s	6820000000ns	1.516s	1516000000ns
5	6.918s	6918000000ns	1.466s	1466000000ns
6	8.408s	8408000000ns	1.502s	1502000000ns
7	7.225s	7225000000ns	1.461s	1461000000ns
8	7.331s	7331000000ns	1.399s	1399000000ns
9	7.022s	7022000000ns	1.415s	1415000000ns
10	8.649s	8649000000ns	1.542s	1542000000ns
MEAN TIME	**7.604s +/- 0.749s**	7604000000ns	**1.473s +/- 0.087s**	1473000000ns

- **Analysis:**

The performance of multi-threaded matrix multiplication was better because it run multiple thread connections. The mean time of multi-threaded program was around 1.473 seconds whereas the mean time of the original program was around 7.604 seconds. There is almost difference of 6 seconds.

2.3 Password cracking using POSIX Threads

- The code is included under the folder Task 2_C_1

- **Table of password cracking using POSIX Single Thread (Two Uppercase and Two Digit Integer)**

REPETITIONS	SECONDS	NANOSECONDS
1	171.121s	171121000000ns
2	168.946s	168946000000ns
3	165.668s	165668000000ns
4	167.853s	167853000000ns
5	188.104s	188104000000ns
6	176.688s	176688000000ns
7	172.389s	172389000000ns
8	169.925s	169925000000ns
9	166.779s	166779000000ns
10	167.154s	167154000000ns
MEAN TIME	**171.463s +/- 6.333s**	171463000000ns
ELAPSED TIME	1,714.285s	1714627000000ns

- **Estimation for the program where the uppercase alphabet letter is increased to 3**

The total time taken to crack the password by the original program which contains two uppercase alphabet letters, and two digits integer elapsed time was about 1714.285 seconds.

Assume the alphabet letter value is 26 (as one alphabet contains 26 letter)

Here, for the two alphabets letter value will be $26^2 = 26 * 26$

and for the integer the value is 100 because there in only one loop in the function which visit from 0 to 99 and find the integer

Therefore,

the previous program run time can be also assumed as $(26^2 * 100)$ which is 1714.285 seconds

Then,

If the number of alphabets will be increased to 3 then one more loop will be added to the function = $26^3 = 26 * 26 * 26$

The total estimation time for the 3 uppercase and 2 integers will be $= (26^2 * 100) * 26$ (Here, 26 is the value of one letter)

$= 1{,}714.285 * 26$ (1,714.285 is the run time of pervious program)

$= 44{,}571.41$

So, the estimated time for the program which contains three upper case letter, and two integers is **44,571.41**

- The code is included under the folder Task 2_C_3

- **Time performance of password cracking using POSIX Single thread (Three Uppercase Letter and Two Integer)**

REPETITIONS	SECONDS	NANOSECONDS
1	4183.860s	4183860000000ns
2	4098.609s	4098609000000ns
3	4011.186s	4011186000000ns
4	3997.939s	3997939000000ns
5	3991.240s	3991240000000ns
6	4002.586s	4002586000000ns
7	4008.709s	4008709000000ns
8	4037.857s	4037857000000ns
9	4066.905s	4066905000000ns
10	4077.920s	4077920000000ns
MEAN TIME	**4047.687s +/- 57.504s**	4047687000000ns
ELAPSED TIME	40476.811s	40476811000000ns

- **Hypothesis and analysis after the run time**

The total time taken to crack the password by the program which contains three uppercase alphabet letters, and two digits integer elapsed time was about 40476.811 seconds. The program ran 11 hours which is 22 times slower than the previous version of the program which contained only two uppercase letter and explored all the possibilities.

The total time estimated earlier was about 44,571.410 which is higher than the program run time.

= Total time estimated – Total run time of program

= 44,571.410 - 40476.811

= 4,094.599 seconds

The encrypted password was SH75 for the first program and SAM62 for three uppercase case programs. The estimation was not accurate but faster than estimated because the alphabets and integer were different and performance was also good so, it ran faster than estimation. The run time and the performance depend upon the loops in the function they run continuously until they find the password and then encrypt them.

- The source-code is included under the folder Task 2_C_5

- **Time performance of password cracking using POSIX Multi thread (Two Uppercase Letter and Two Integer)**

REPETITIONS	SECONDS	NANOSECONDS
1	139.249s	139249000000ns
2	141.121s	141121000000ns
3	141.598s	141598000000ns
4	141.736s	141736000000ns
5	141.639s	141639000000ns
6	140.409s	140409000000ns
7	140.676s	140676000000ns
8	141.741s	141741000000ns
9	140.903s	140903000000ns
10	141.141s	141141000000ns
MEAN TIME	**141.021s +/- 0.736s**	141021000000ns
ELAPSED TIME	1410.213s	1410213000000ns

- **Comparison between the single threaded and multi-threaded version of password cracking (Two Uppercase and Two Integer)**

REPETITIONS	SINGLE THREADED VERSION		MULTI-THREADED VERSION	
	SECONDS	NANOSECONDS	SECONDS	NANOSECONDS
1	171.121s	171121000000ns	139.249s	139249000000ns
2	168.946s	168946000000ns	141.121s	141121000000ns
3	165.668s	165668000000ns	141.598s	141598000000ns
4	167.853s	167853000000ns	141.736s	141736000000ns
5	188.104s	188104000000ns	141.639s	141639000000ns
6	176.688s	176688000000ns	140.409s	140409000000ns
7	172.389s	172389000000ns	140.676s	140676000000ns
8	169.925s	169925000000ns	141.741s	141741000000ns
9	166.779s	166779000000ns	140.903s	140903000000ns
10	167.154s	167154000000ns	141.141s	141141000000ns
MEAN TIME	**171.463s +/- 6.333s**	171463000000ns	**141.021s +/- 0.736s**	141021000000ns
ELAPSED TIME	1714.627s	1714627000000ns	1410.213s	1410213000000ns

3 Applications of Password Cracking and Image Blurring using HPC-based CUDA System

3.1 Password Cracking using CUDA

- The source-code is included under the folder name Task 3_1
- **Table of time comparison between the original and the CUDA version of password cracking**

REPETITIONS	CUDA VERSION		MULTI-THREAD VERSION		SINGLE THREADED VERSION	
	SECONDS	NANOSECONDS	SECONDS	NANOSECONDS	SECONDS	NANOSECONDS
1	0.098s	98000000ns	139.249s	139249000000ns	171.121s	171121000000ns
2	0.093s	93000000ns	141.121s	141121000000ns	168.946s	168946000000ns
3	0.092s	92000000ns	141.598s	141598000000ns	165.668s	165668000000ns
4	0.098s	98000000ns	141.736s	141736000000ns	167.853s	167853000000ns
5	0.106s	106000000ns	141.639s	141639000000ns	188.104s	188104000000ns
6	0.090s	90000000ns	140.409s	140409000000ns	176.688s	176688000000ns
7	0.101s	101000000ns	140.676s	140676000000ns	172.389s	172389000000ns
8	0.104s	104000000ns	141.741s	141741000000ns	169.925s	169925000000ns
9	0.110s	110000000ns	140.903s	140903000000ns	166.779s	166779000000ns
10	0.096s	96000000ns	141.141s	141141000000ns	167.154s	167154000000ns
MEAN TIME	**0.0988s**	98800000ns	**141.021s +/- 0.736s**	141021000000ns	**171.463s +/- 6.333s**	171463000000ns
ELAPSED TIME	0.988s	988000000ns	1410.213s	1410213000000ns	1714.627s	1714627000000ns

- **Analysis and evaluation:**

The table above the time performance and comparison between two password cracking methods CUDA and POSIX. POSIX using single thread and multi thread versions for cracking the password. As we all know CUDA is faster than any other method and the table also shows the performance. The meantime of CUDA was about 0.0988 seconds whereas the multi trade took 141.021 seconds and single thread took 171.463 seconds. There is a huge difference of time. CUDA version ran about 150 times faster than POSIX. The password to be cracked was also the same in all three version of program which is SH75.the password contain two uppercase alphabet letters and two-digit integers.

However, the time performance depends upon the performance of PC because the CUDA can run only on the GPU and GPU is faster than normal CPU. As the program of POSIX uses single and multiple thread to crack the password, they must visit multiple connections so, it took more time. Multi thread was faster than single threaded program. The single thread and multi thread program run multiple loops so it takes more time and there is not much difference between them. The CUDA program provides better performance and is best for password cracking.

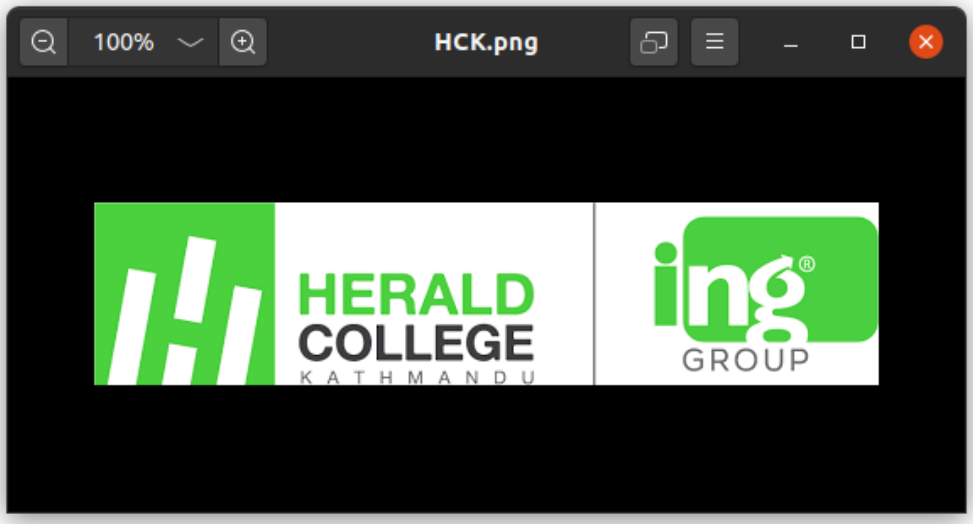
3.2 Image blur using multi dimension Gaussian matrices

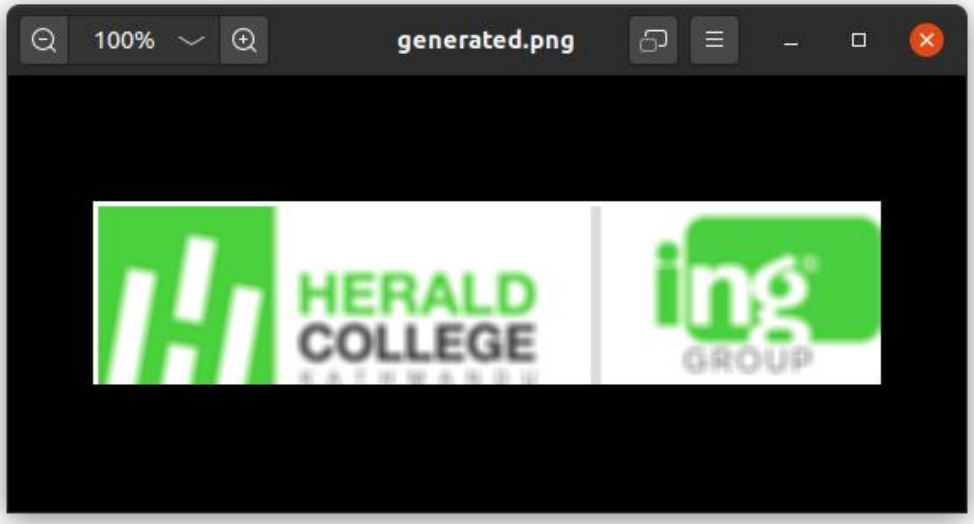
- The code and library are included under the folder name Task 3_2

- **Time performance of image blur using gaussian matrix**

REPETITIONS	SECONDS	NANOSECONDS
1	0.244s	244000000ns
2	0.020s	20000000ns
3	0.020s	20000000ns
4	0.020s	20000000ns
5	0.020s	20000000ns
6	0.020s	20000000ns
7	0.020s	20000000ns
8	0.020s	20000000ns
9	0.020s	20000000ns
10	0.021s	21000000ns
MEAN TIME	**0.042s +/- 0.067s**	42000000ns

- **Screenshots of comparison between the original image and the blurred image**





- **Analysis:**

In above program we used 3x3 multi pixel processing Gaussian matrix to blur the original image (.png) which has the RGB values. The matrix uses gaussian function which calculates the transformation to apply to each pixel in the image. Gaussian blur matrix implements a weighted average around the pixel. The complexity can be defined as O (N*r*r) where n is the total number of pixels. The time performance was also good because of the kernel function in the program. The screenshots above show the result the original image as well as the blurred image. The program took less than a second to execute. The total number of test count in the program contain is ten where after total test is performed the mean average time is calculated. The mean average time of this program was 0.042 seconds. However, the performance can be improved by using different pixels. The CUDA program performed well.

4 Verbose Repository Log

commit 654aee6bd068d24a0dc2b4e7fc2f6a048abcd18a
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Sat Dec 26 12:44:01 2020 +0545

Final Commit

commit 6335c4c4ed90009553e5e106c3eba778cd170e5c
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Sat Dec 26 02:08:47 2020 +0545

Cuda Password Crack

commit 87e406d5e141d1d8880eb3515229acbe6cf26c04
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Sat Dec 26 00:31:41 2020 +0545

Screenshots

commit c71924bd0ea42bfd84261e20842cdc4776999c35
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Fri Dec 25 16:18:45 2020 +0545

Test

commit a6a84748eebd9b961e771a755282a2863bc61fe8
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Fri Dec 25 00:19:41 2020 +0545

Matrix Multiplication

commit ace58baf87a50656b809a8646c0d249592620c6f
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Thu Dec 24 22:30:33 2020 +0545

Initial Commit

commit 77a94524f501b27fe200357b8b4659e077b4b6ca
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Thu Dec 24 18:50:09 2020 +0545

Image Blur Gaussain

commit 3368a4044011b0551a028e9cd5ba947bb2bddccb
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Thu Dec 24 14:46:16 2020 +0545

POSIX Password Crack

commit 9b4478c60bedb3ea082b6151b43f5459b33409a9

Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Tue Dec 22 15:30:37 2020 +0545

Task 1 5

commit c4059102acff4f7f2178dc7e5f9a477f45b1e636
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Sun Dec 20 19:30:51 2020 +0545

Started POSIX

commit 6ddfa5e43a5462ac6c00b4cad52c40b7850dd00a
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Fri Dec 4 14:35:15 2020 +0545

Second Commit

commit c2f1f0e7b1db0f65fca3fb564bbde2886ce4cba1
Author: Samir Husen <saw.mendes.meer@gmail.com>
Date: Fri Dec 4 14:23:41 2020 +0545

first commit