Developing Soft and Parallel Programming Skills Using Project-Based Learning

Semester: Fall 2018

Group Name: Dunder Mifflin

Members: Anli Ji, Brian Cabau, Jimmy Petit Homeis, Manuel Lipo, Riyazh Dholakia

Task 1 Planning and Scheduling:

Name's	Email	Task	Duration (hours)	Dependency	Due date	Note
Riyazh Dholakia **(Coordinator)**	rdholakia1@student.gsu.edu	Team coordinator, building task assignment table, turning final written report, working on parallel architecture and programming on Pi, GitHub contribution and Repo/Project Setup	3 hours	Table must be done before next meeting on 10/12. Work on pi before answering some of the questions.	10/25/18	Includes parallel programming skills and GitHub
Anli Ji	aji1@student.gsu.edu	Writing and formatting the lab report, working on parallel architecture and programming on Pi, adding explanation to report, helping with GitHub Project	3 hours	Lab report written during group meeting. Work on pi before answering some of the questions.	10/24/18	Document the lab report as we're all working on it
Jimmy Petit Homeis	jpetithomeis1@student.gsu.edu	Raspberry pi re-setup, video editing (uploading video), final written report, filming, working on parallel architecture and programming on Pi, GitHub contribution	3 hours	Pi setup must be complete before group work can begin. Video must be filmed before editing.	10/24/18	Includes parallel programming skills
Manuel Lipo	mlipo1@student.gsu.edu	Raspberry pi re-setup and parallel architecture and programming, GitHub contribution	2 hours	Raspberry pi setup and programming	10/24/18	Document the lab report as we're all working on it
Brian Cabau	bcabau1@student.gsu.edu	GitHub, Slack invitation, final written report, working on parallel architecture and programming on Pi, GitHub contribution	3 hours	GitHub must be ready by 9/24. Pi must be completed before working on Lab Report.	10/24/18	Includes parallel programming skills and basics

Task 2 Parallel Architecture and Programming Skills:

a) Foundation

Define the following:
Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words)

Task is the instructions that must be executed by a processor; it's like a goal that is accomplished in written steps by the processor.
Pipelining splits task into distinct parts so that the task can be done through parallel programming. Input flows through the pipeline to be executed by different cores of the processor.
Shared memory means that the processor(s) or cores all utilize the same memory. It creates dependencies, and we saw the problem it can cause from the second assignment where multiple threads were printing the same output as another.
Communications are the way processors exchange data in parallel; it is the manner of doing so.
Synchronization is related to communications, it should do with timing tasks in parallel so they may wait for other tasks, or continue when a task reaches a point.

Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.

Single data: is a single Data stream that is fed into multiple processing units.
Multiple data: is where every processor may be working with a different data stream.

Single instruction, single data:
A non-parallel computer where the Single instruction is one instruction that has been acted on the CPU during one clock cycle. Single data, is where only one data stream is being used as input during any clock cycle.

Single instruction, multiple data:
A parallel computer where all processing unit can execute the same instruction at any given clock cycle. Multiple Data, is where each processing unit can operate on a different data element

Multiple instruction, single data:
A parallel computer where each processing unit operates on the data independently via separate instructions streams

Multiple instruction, multiple data:
A parallel computer that every processor may be executing a different instruction stream

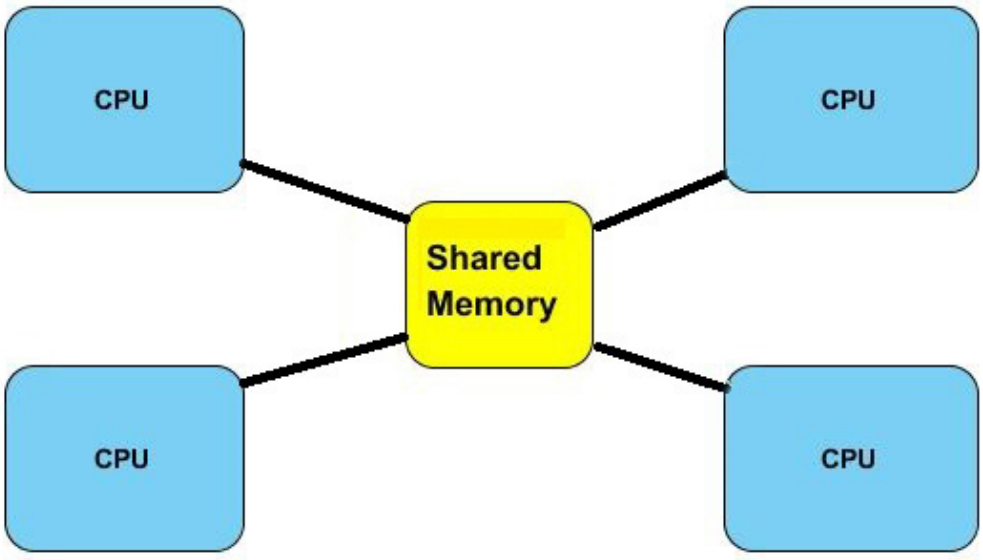
What are the Parallel Programming Models?
Shared Memory (without threads)
Threads
Distributed Memory / Message Passing
Data Parallel
Hybrid
Single Program Multiple Data (SPMD)
Multiple Program Multiple Data (MPMD)

List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?

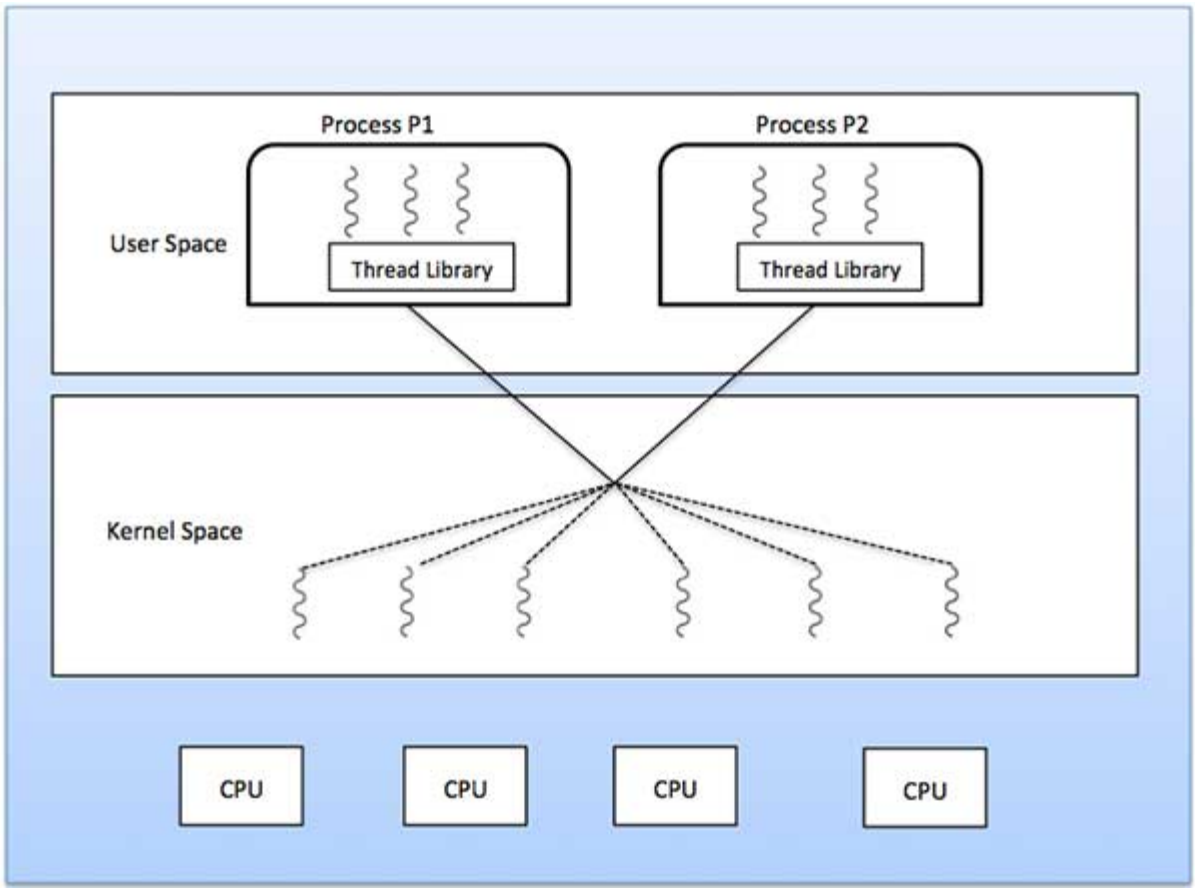
There're two types of shared memory architectures based on their memory access time known as UMA and NUMA. UMA is defined as uniform memory access while NUMA is defined as non-uniform memory access. In UMA, the processors are identical and each of them has equal access and access time to memory. It mostly represented today by Symmetric Multiprocessor (SMP) machines. UMA sometimes also called Cache Coherent UMA. It means all the processors will know about the update if any single one of them makes some updates. In contrast, not all processors will have equal access time to all memories in NUMA. The memory access across link is comparably slower than in UMA. Also, NUMA is made by physically linking two or more SMP and each one of these SMP can have direct access to memory of other SMP. The type that OpenMP used is UMA since OpenMP is an API that supports shared memory multiprocessing in multi-platform.

Compare Shared Memory Model with Threads Model? (in your own words and show pictures)

In Shared Memory Model, all the processors can have equal access to the shared memory. The processes/tasks can read or write asynchronously by sharing a common address in memory. They use various mechanisms like locks/semaphores to control the access in shared memory in a way to avoid race conditions and deadlocks. However, this kind of model will have difficulty in understanding and managing data locality.



In a thread model, a single upper process can have multiple subsets with concurrent execution paths. Each of the thread has their local data and shares the whole program resources from memory.



What is Parallel Programming? (in your own word)

Parallel programming is a software that have been designed to write coding in a sequential way which is executed in a single processor. The code is split into different series of instruction and each of those instructions are executed one after another.

What is system on chip (SoC)? Does Raspberry PI use system on SoC?

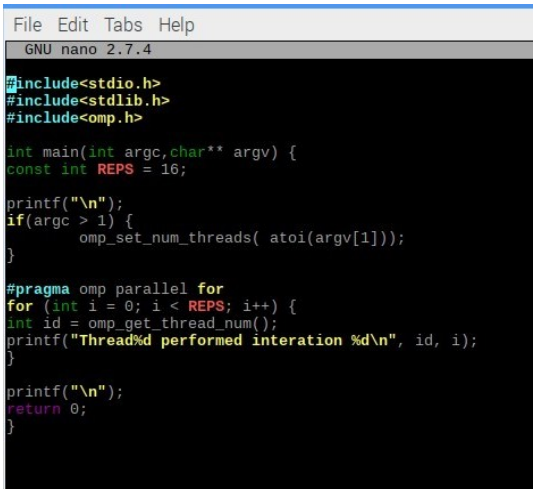
SoC is basically multiple components on one chip such as CPU, memory (RAM and ROM), input/output ports, possible modem, and secondary storage (example hard disk drive). SoC's are compact and take less power than multi-chips, these are now very common especially with mobile devices. Yes, our Raspberry Pi is on SoC.

Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.

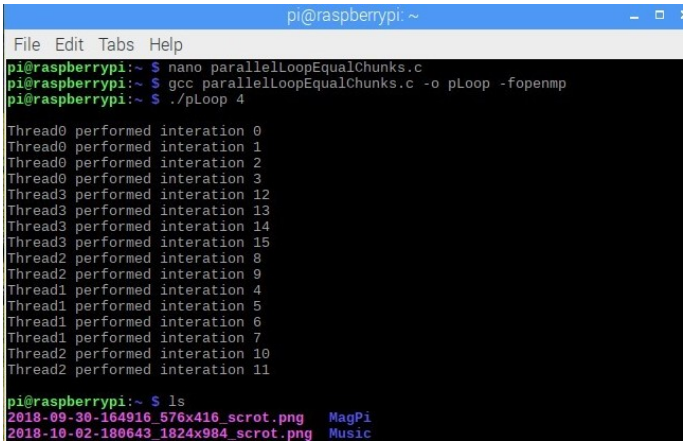
There a quite a few benefits of SoC's. Benefits of SoC's: compact (require less space and light weight), high performance due to more circuits on chip, lower cost, high speed processor and memory, ideally should have greater security, and low power requirements. When the components is separate more space, heavier, more power required. Good part is if one component like GPU goes bad it can be replaced rather than the whole SoC.

b) Parallel Programming Basics

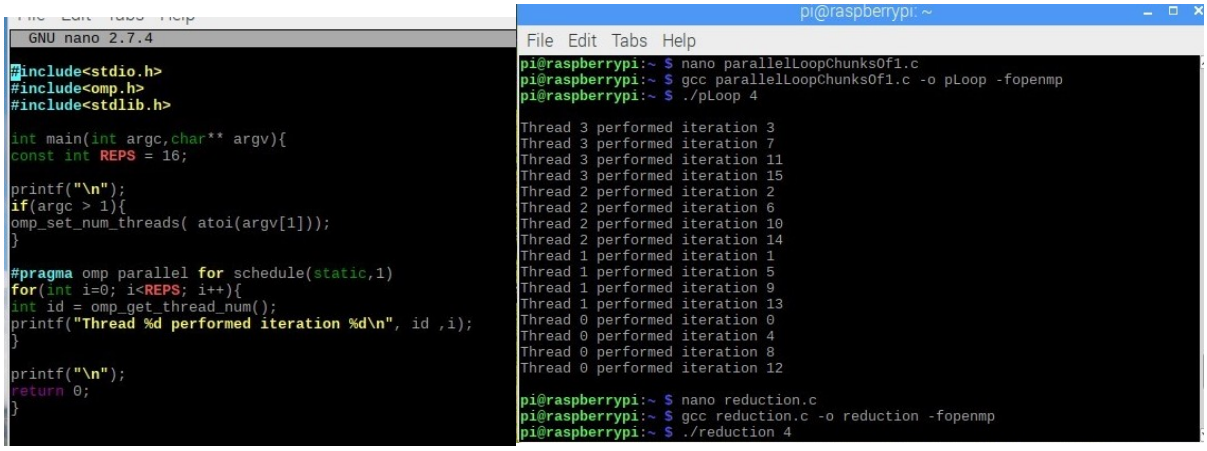
After opening the terminal and creating the file in nano, we entered the code that would iterate a loop between forked threads. This is interesting because there is more than one way to accomplish this, known as the data decomposition pattern.



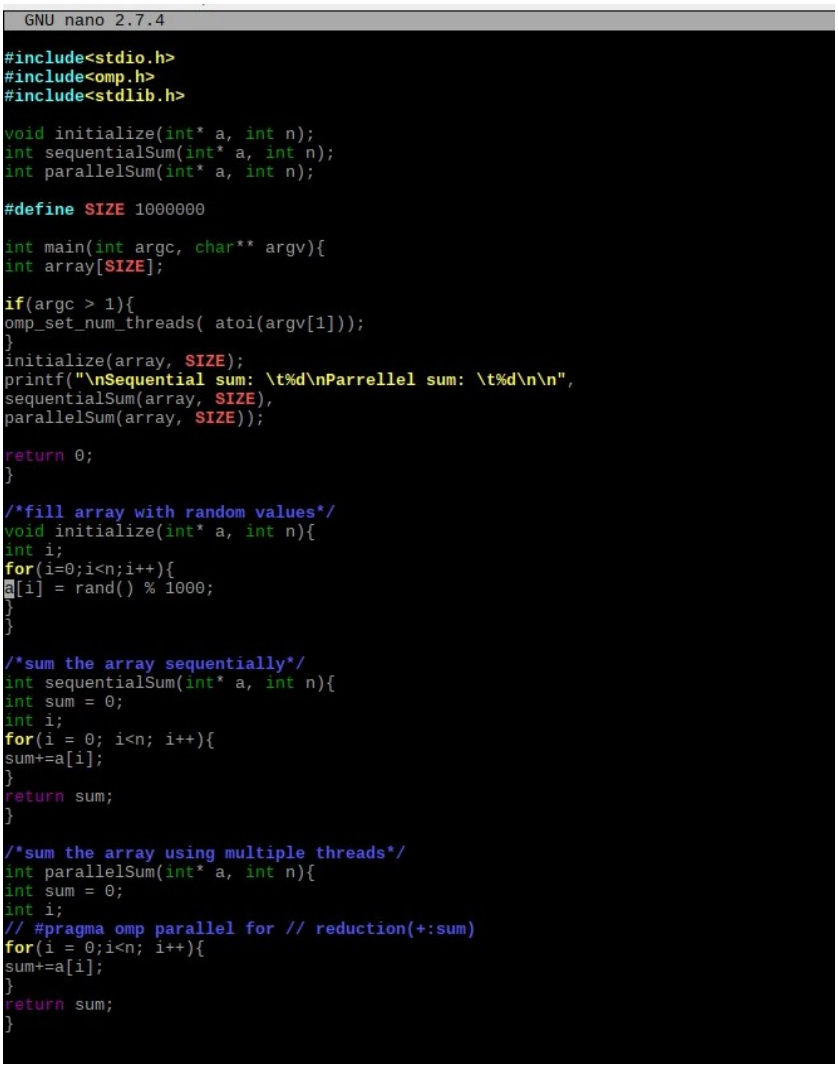
After typing the code, the file had to be complied to create an executable file. When executing the file, we used the argument 4 to signify that the loop should forked into 4 threads. The output shows that each of the 4 threads completed 4 iterations, equaling the 16 iterations of the loop. It is interesting how the output does not show a sequential order; thread 3 finishes before thread 2 for example.



The next C program is intended to do the same thing, but with a slightly different implementation. The difference in this code is that it is iterated through single chunks; each thread completes one iteration at a time before doing the next. This is what the change in line 10 does. It makes each loop do one iteration. What is interesting is that each thread still completes the same amount of work since it is done statically.



The next C code demonstrates an accumulator that works in parallel, typing the code, we can see that it is not running in parallel until we uncomment line 39. In the first case by typing the code exactly as is, the two methods that are executed show the same value. This is the result we want, but it should be done through parallel programming.



```
^[[A^[[Api@raspberr./reduction

Sequential sum:         499562283
Parrellel sum:  499562283

pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parrellel sum:  499562283

pi@raspberrypi:~ $ ./reduction 5

Sequential sum:         499562283
Parrellel sum:  499562283

pi@raspberrypi:~ $
```

Uncommenting #pragma on line 39 doesn't produce the correct output. The results of sequentialSum and parallelSum do not match. According to the document, this is because the accumulator, sum, should be private. When reduction(+:sum) is commented, this is not the case. It is interesting to see how the result changes, and how much control the user has over the low-level components of the CPU.

```
/*sum the array sequentially*/
int sequentialSum(int* a, int n){
int sum = 0;
int i;
for(i = 0; i<n; i++){
sum+=a[i];
}
return sum;
}

/*sum the array using multiple threads*/
int parallelSum(int* a, int n){
int sum = 0;
int i;
#pragma omp parallel for // reduction(+:sum)
for(i = 0;i<n; i++){
sum+=a[i];
}
return sum;
}

^G Get Help      ^O Write Out     ^W Where Is
^X Exit          ^R Read File     ^\ Replace
```

```
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
^[[Api@raspberrypi:./reduction

Sequential sum:         499562283
Parrellel sum:  155034935

pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parrellel sum:  156107963

pi@raspberrypi:~ $ ./reduction 5

Sequential sum:         499562283
Parrellel sum:  224348098

pi@raspberrypi:~ $
```

But uncommenting the rest of the line once again produces the correct output. This time however, it is done using parallel programming. Uncommenting reduction on line 39 shows the dependency on the threads computation of their sums, because the accumulator sum should be private. If it is not private, the program will not produce the correct output.

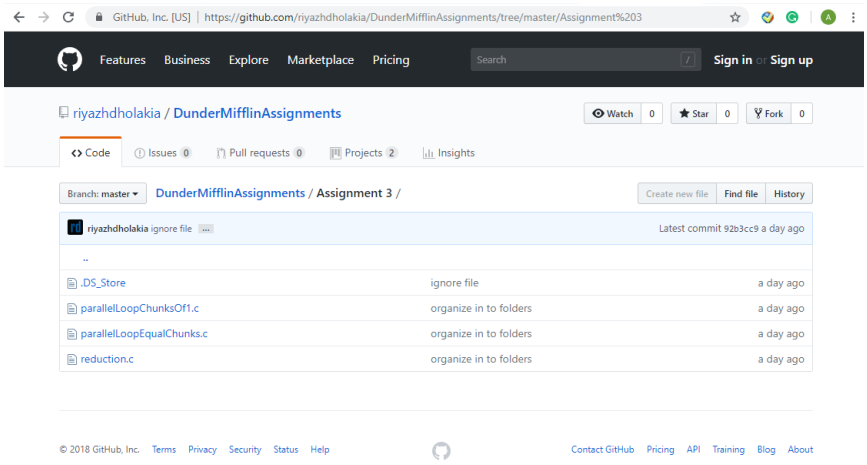
```
for(i = 0; i<n; i++){
sum+=a[i];
}
return sum;
}

/*sum the array using multiple threads*/
int parallelSum(int* a, int n){
int sum = 0;
int i;
 #pragma omp parallel for  reduction(+:sum)
for(i = 0;i<n; i++){
sum+=a[i];
}
return sum;
}

^G Get Help      ^O Write Out     ^W Where Is
^X Exit          ^R Read File     ^\ Replace
```

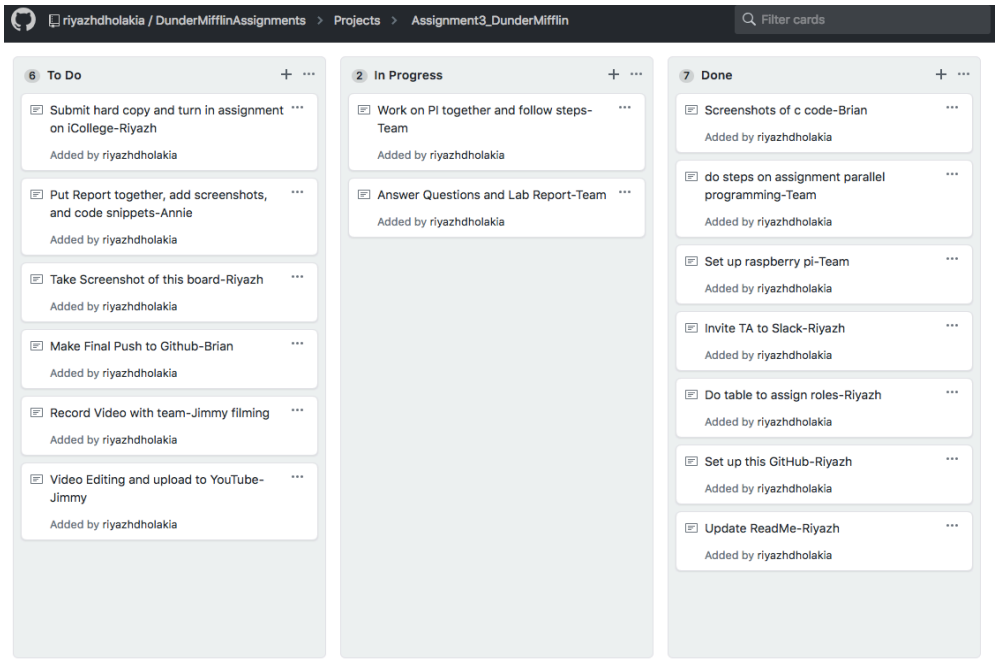
```
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction

Sequential sum:         499562283
Parrellel sum:  499562283

pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parrellel sum:  499562283

pi@raspberrypi:~ $ ./reduction 5

Sequential sum:         499562283
Parrellel sum:  499562283

pi@raspberrypi:~ $ ./reduction 2

Sequential sum:         499562283
Parrellel sum:  499562283

pi@raspberrypi:~ $
```

Task 3 Appendix:

Slack Invite Link:
https://join.slack.com/t/dundermifflingang/shared_invite/enQtNDM0MzAzNTc0ODUyLTQwMDJlNDMyY2I0M2Q2MjY3MzA5ZjdiM2FmOWRiZjA0ZThmOGNkNGY1YWM4NGQ4ZDBmODllMWI5NjFkZDljYWU

GitHub Project Screen Shot:





YouTube Channel Link:

https://www.youtube.com/channel/UCbAorx7CYVlyDs7_38edPRg

Video Link: https://youtu.be/4bX-4r2YMeA