Developing Soft and Parallel Programming Skills Using Project-Based Learning
Fall 2019
Team CHABU
Chris Lavey, Harsh Jivani, Anggiela Yupanqui, Binh Ha Duy Le, Ugonma Nnakwe

1. Planning and Scheduling

Work Breakdown

Assignee Name	Email	Task	Duration (Hours)	Dependency	Due Date	Note
Harsh Jivani	hjivani1@student.gsu.edu	Slack, Parallel Programming	3 hours	None	11/5/19	100% grade
Chris Lavey	clavey@student.gsu.edu	Parallel Programming, Technical writing (getting the report ready) as described in the assignment	6 hours	Slack, Github, Parallel Programming, ARM Assembly Programming	11/14/19	100% grade
Anggiela Yupanqui	ayupanquirojas1@student.gsu.edu	GitHub, ARM Assembly Programming	3 Hours	None	11/4/19	100% grade
Ugonma Nnakwe **(coordinator)**	unnakwe1@student.gsu.edu	Video editing, ARM Assembly Programming	5 Hours	Video recording (11/8 at 11:00AM)	11/14/19	100% grade
Binh Le	ble8@student.gsu.edu	Parallel Programming	3 hours	None	11/5/19	100% grade

2. Parallel Programming Skills

2.1. Foundation

2.1.1. What is race condition?

Race condition is the behavior of a system where the output is directly affected by another event whose timing or sequence is uncontrollable. Because the events are meant to run sequentially, this makes it difficult to debug as a bug may not be apparent if the events don't run in order.

2.1.2. Why race condition is difficult to reproduce and debug?

A race condition can be difficult to reproduce because the end result is dependent on the timing between the threads. When debugging, the problems created can disappear, making the debugging process more difficult. To minimize this issue, it is better to work on a more efficient software design than to try and reproduce and debug a race condition later.

2.1.3. How can it be fixed? Provide an example from Project A3

To avoid a race condition, it is important to plan your code and develop a proper code design for the program. Below is an example from spmd2.c where one of the thread's IDs appears more than once:

```
 GNU nano 2.3.1                File: spmd2.c

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc,char** argv) {
//int id, numThreads;
printf("\n");
if (argc > 1) {
omp_set_num_threads( atoi(argv[1]));
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
printf("Hello from thread %d of %d\n",id,numThreads);
}
printf("\n");
return 0;
}
```

The output of this program would be similar to this:

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4

Our goal here was to have a unique thread ID for every thread, and because of an error in the code this was not always guaranteed. When analyzing our code, we realized that the variable **id** was declared outside of the parallel function, resulting in it being shared among all the threads. To correct this issue, we commented out the initial declarations and then declared these variables inside the parallel function.

2.1.4. Summarize the Parallel Programming Patterns section

There are two important patterns used in parallel programming: strategies and concurrent execution mechanisms. Strategies that implement parallel programming can be further divided into two subcategories: implementation strategies and algorithmic strategies. Implementation strategies will determine how the tasks of a program are going to be processed, while algorithmic strategies determine how and when these tasks should be executed. Concurrent execution mechanisms handle the hardware. Concurrent execution mechanisms can be divided into two major subcategories: process/thread control patterns and coordination patterns. Process/thread control patterns control which thread will be processed and when this will happen at runtime. Coordination patterns allocate processing units to handle concurrently running tasks to complete parallel computations. There is also a third implementation, which is a hybrid computation that uses a combination of strategies and concurrent execution mechanisms. This pattern uses both OpenMP and Message Passing Interface.

2.1.5. Collective Synchronization vs. Collective Communication

In collective synchronization, a barrier is created such that a group of threads or processes must stop at this point and cannot proceed until all the other threads/processes have reached it. This uses the function **MPI_Barrier()**.

In collective communication, a single process, called the root process, collects data from the other processes in a group of threads and performs an operation such that it produces a single value. This uses the function **MPI_Reduce()**.

2.1.6. Master-Worker vs. Fork Join

In a master-worker pattern, a main process is divided into small chunks which are handled by several worker processes. In a fork-join pattern, multiple lightweight processes are executed in parallel.

2.1.7. Where can we find parallelism in programming?

Parallelism is used as a means to have a program be considerably more efficient. This is done by making several computations in parallel. To code a program in this manner, it requires hardware that contains multiple processing units. A computer with a multicore processor is capable of running a parallel program.

2.1.8. What is dependency and what are its types?

A dependency arises when one operation requires an earlier operation to return a result before it can be performed. There are three examples of dependency, listed below with examples:

1. True dependence:
   S1: a = b + c;
   S2: d = a + 2;
2. Output dependence:
   S1: a = b + c;
   S2: a = d / 2;
3. Anti-dependence:
   S1: a = b + c;
   S2: b = d / 2;

2.1.9. When a statement is dependent/independent (with examples)

Statement S2 is dependent on statement S1 when the order of their execution affects the outcome of the computation. This can be seen in the example below:

S1: a = 100;
S2: b = a; //when a changes in S1, b in S2 changes to reflect this

Statement S2 is independent of statement S1 when their order of execution does not matter. This is shown in the example below:

S1: a = 100;
S2: b = 100; //when a changes in S1, b does not change to reflect it

2.1.10. When can two statements be executed in parallel?

Two statements S1 and S2 can execute in parallel iff S1 is independent of S2 and vice versa.

2.1.11. How can dependency be removed?

Dependency can be removed in two ways: rearranging statements, and eliminating statements.

2.1.12. How do we compute dependency for the two loops below and what is the type(s) of dependency?

```
for (i=0; i<100; i++)
   S1: a[i] = i;
```

```
for (i=0; i<100; i++) {
   S1: a[i] = i;
   S2: b[i] = 2*i;
}
```

for (i=0; i<100; i++)
** S1: a[i] = i;**

$S1^0$ $S1^1$ $S1^2$ … $S1^{99}$

for (i=0; i<100; i++) {
** S1: a[i] = i;**
** S2: b[i] = 2*i;**
}

$S1^0$ $S1^1$ $S1^2$ … $S1^{99}$

□ □ □ □

$S2^0$ $S2^1$ $S2^2$ … $S2^{99}$

These two loops both have true dependencies.

2.2. Parallel Programming Basics

2.2.1. trap-notworking.c and trap-working.c

Our first task required us to copy two programs, **trap-notworking.c** and **trap-working.c** and examine the two to better understand how reduction works. Using the **nano** terminal, we wrote the following for **trap-notworking.c**:

```c
#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP


/* Demo program for OpenMP: computes trapezoidal approximation to an integral*/

const double pi = 3.14159265358979323846264338307 9;

int main(int argc, char** argv) {
/* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a) / n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;

double f(double x);

/* parse command-line arg for number of threads */
if (argc > 1) {
threadcnt = atoi(argv[1]);
}

#ifdef _OPENMP
omp_set_num_threads( threadcnt );
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif

integral = (f(a) + f(b))/2.0;
int i;

#pragma omp parallel for private(i) shared (a, n, h, integral)
for(i = 1; i < n; i++) {
integral += f(a+i*h);
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n", n);
printf("%f to %f is %f\n", a,b,integral);
}

double f(double x) {
return sin(x);
}
```

And below is what we wrote for **trap-working.c**:

```c
#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP


/* Demo program for OpenMP: computes trapezoidal approximation to an integral*/

const double pi = 3.14159265358979323846264338307 9;

int main(int argc, char** argv) {
/* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a) / n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;

double f(double x);

/* parse command-line arg for number of threads */
if (argc > 1) {
threadcnt = atoi(argv[1]);
}

#ifdef _OPENMP
omp_set_num_threads( threadcnt );
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif

integral = (f(a) + f(b))/2.0;
int i;

#pragma omp parallel for \
private(i) shared (a, n, h) reduction(+: integral)
for(i = 1; i < n; i++) {
integral += f(a+i*h);
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n", n);
printf("%f to %f is %f\n", a,b,integral);
}

double f(double x) {
return sin(x);
}
```

The key difference between these two is line 37, where we have an accumulator variable called **integral**. In order for this to be calculated properly, we must use the reduction clause for that variable. We used **gcc** to compile the two programs, and below is the output once we ran these programs:

```
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.422894
pi@raspberrypi:~ $ ./trap-working 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~ $
```

The goal of these programs was identical: to approximate the integral of sin(x) from 0 to pi using trapezoids. The integral of sin(x) from 0 to pi is 2, and by examining the two results it is clear that **trap-notworking** does not produce this.

This is because it doesn't use reduction on the value of **integral**, resulting in it not being calculated properly. By using reduction in **trap-working**, we get the value we desired. This is because the reduction function collects the data calculated on each thread and performs an operation in order for the value to be stored in one process. In this case, the values were accumulated into the variable **integral**.

2.2.2. barrier.c

Our second task required us to write the program **barrier.c** and correct it as necessary to gain a better understanding of our barriers work in parallel programming. Using the **nano** terminal, we wrote the code seen below for **barrier.c**:

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {
        printf("\n");
        if(argc > 1) {
         omp_set_num_threads(atoi(argv[1]));
        }

        #pragma omp parallel
        {
          int id = omp_get_thread_num();
          int numThreads = omp_get_num_threads();
          printf("Thread %d of %d is BEFORE the barrier. \n", id,numThreads);

//        #pragma omp barrier

          printf("Thread %d of %d is AFTER the barrier. \n", id,numThreads);
        }

        printf("\n");
        return 0;
}
```

In the screenshot above, the variables **id** and **numThreads** are initialized and declared inside the parallel function, where they are used to print the following two strings for each thread (the first %d is **id** and the second is **numThreads)**:

Thread %d of %d is BEFORE the barrier.

Thread %d of %d is AFTER the barrier.

The goal is to have all strings including **ABOVE** to be before all strings including **AFTER**. After compiling our program using the **gcc** compiler, this was the output:

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

Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
```

This output had each thread print their strings sequentially, rather than having each thread handle their strings before and after the barrier. By uncommenting the line **#pragma omp barrier**, this was the result:

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

Thread 0 of 4 is BEFORE the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
```

By using the **barrier** function, we were able to get the desired output. This is because when using a barrier, the threads have to wait on every task before the barrier to be finished before they can move on to tasks after it.

2.2.3. masterWorker.c

Our final task required us to write the program **masterWorker.c** and correct it as necessary in order to gain a better understanding of the master-worker method of implementation. Using the **nano** terminal, we wrote the below code:

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv) {
        printf("\n");
        if(argc > 1) {
          omp_set_num_threads(atoi(argv[1]));
        }

//      #pragma omp parallel

        {
          int id = omp_get_thread_num();
          int numThreads = omp_get_num_threads();

          if(id == 0) {
            printf("Greetings from the mastr, #%d of %d threads\n", id, numThreads);

          } else {
                printf("Greetings from a worker, #%d of %d threads\n", id,numThreads);
          }
        }

        printf("\n");

        return 0;
```

The goal of this program is to have each thread print a string based on whether it is the master thread or a worker thread. If it is the master thread, it should print "**Greeting from the master, #%d of %d threads**", and if it is a worker thread then it should print "**Greetings from a worker, #%d of %d threads**". We compiled our program using the **gcc** compiler, and the output of the program can be seen below:

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

Greetings from the mastr, #0 of 1 threads
```

This output only displays one string, which comes from the master thread. This is because the program has the parallel function commented out. When uncommenting, recompiling, and re-running, this is the result:

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

Greetings from the mastr, #0 of 4 threads
Greetings from a worker, #3 of 4 threads
Greetings from a worker, #1 of 4 threads
Greetings from a worker, #2 of 4 threads
```

When running **masterWorker** this time, we see that there is a single master thread's string and the strings of three worker threads. This is because by default a parallel program in ARM uses 4 threads. The first thread is designated as the master thread, with each subsequent thread being a worker.

3. ARM Assembly Programming

3.1. Part 1

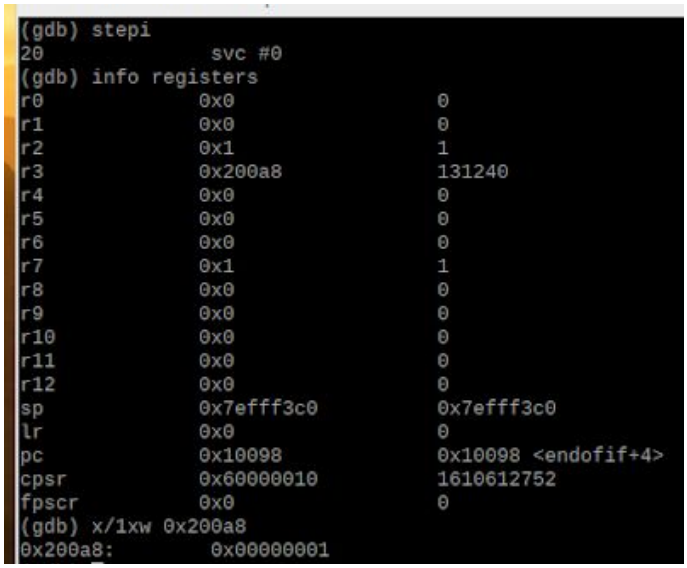
Our first task was to write the program **fourth.s**. Using the **nano** terminal, we wrote the following code:

```
.section .data
x: .word 0
y: .word 0
.section .text
.globl _start
_start:
        ldr r1, =x
        ldr r1, [r1]

        cmp r1,#0
        beq thenpart
        b endofif

thenpart:
        mov r2, #1
        ldr r3, =y
        str r2, [r3]
endofif:
        mov r7, #1
        svc #0
        .end
```

This program shows how to do a simple if statement in ARM. It begins by loading the value of **x** into **r1**, then compares it to 0 using **cmp**. If **r1** equals 0, then the program jumps to **thenpart** using **beq** (branch if equal). If **r1** does not equal 0, then the program jumps to **endofif** using **b** (branch). Inside the if statement (after **thenpart** but before **endofif**), it loads **r2** with the value of 1, then stores this value in **y**.

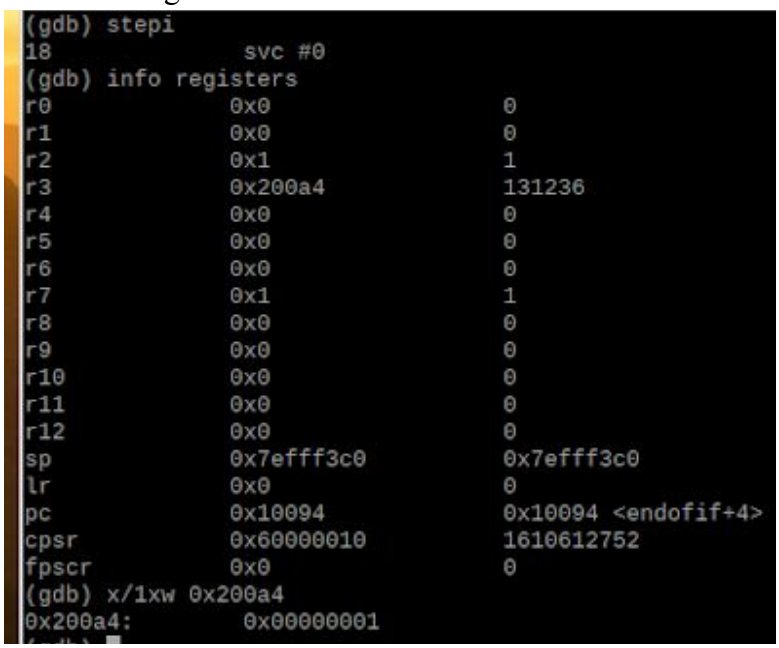
We assembled our program using the **as** command, then linked it using **ld**. We then used the **gdb** debugger to analyze our program. We set a breakpoint in the program using the **b** command, then use the **stepi** command to step through our program one line

at a time until we reach the line whose output we're interested in. We're interested in the line where we store the final value of **y**, so we did this until we reached the line we were interested in, then used the command **info registers** to display the values stored in the registers. From this menu, we were able to get the address for that line and used **x/1xw** to examine the memory at that location in hexadecimal format. This can be seen below.



3.2. Part 2

Our second task was to modify the program for part 1 to improve efficiency. We can do this by only having one jump conditional instead of two. We can remove the **beq** line and change **b** to **bnq**. When we do this, the result will be the same as in part 1, but will be achieved in a much more efficient manner. When examining the registers, we can see that the Z flag is set to 1.



3.3. Part 3

Our final task was to write a program called **ControlStructure1.s** that performed the following operation:

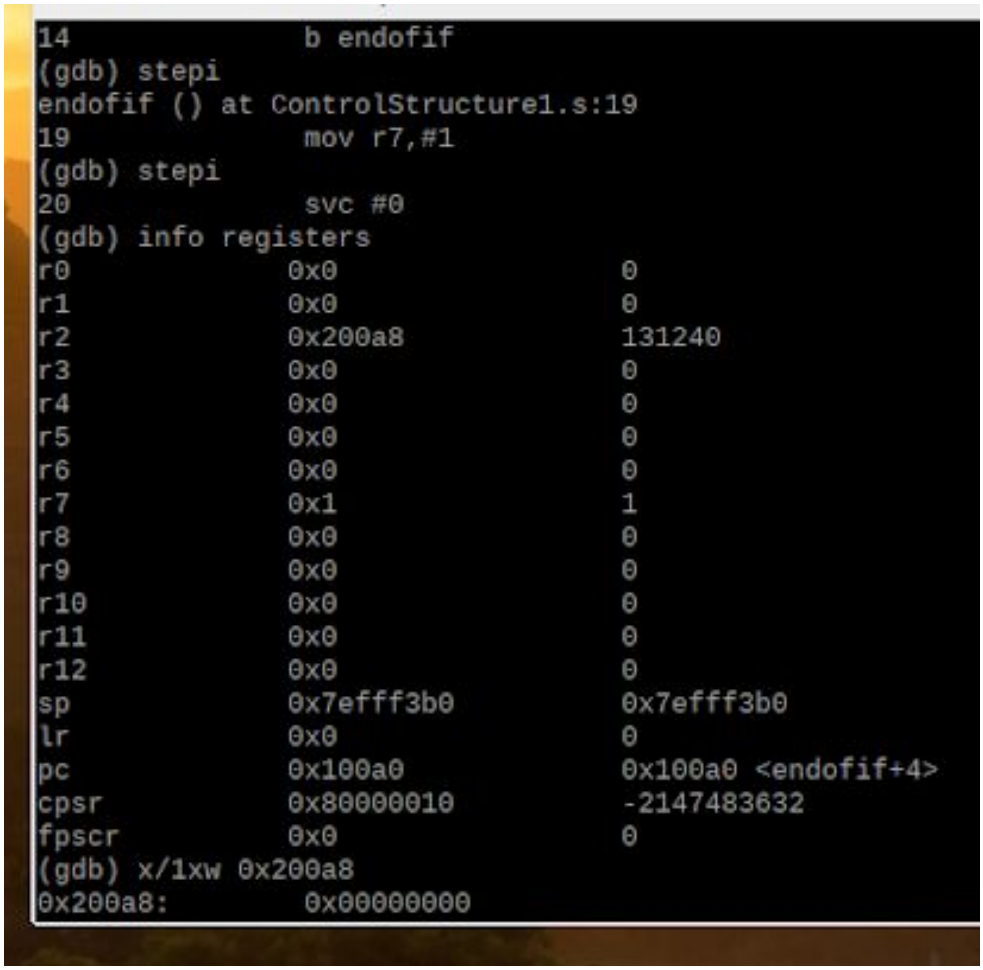
if X<=3
X-=1
else
X-=2

Below is our solution:

```
.section .data
x: .word  1
.section .text
.globl _start
_start:
        ldr r1,=x
        ldr r1,[r1]
        ldr r2, =x
        cmp r1,#3

        bgt next
        sub r1,#1
        str r1,[r2]
        b endofif
next:
        sub r1,#2
        str r1,[r2]
endofif:
        mov r7,#1
        svc #0
        .end
```

This begins by loading **r1** with the value of **x** and **r2** with the address of **x**. We then used **cmp** to compare **r1** to 3. We then used **bgt** to jump to **next** if the value stored in **r1** is greater than 3 (this handles the else statement). If the value of **r1** is less than 3, then we decrement **r1** and store the new value into **x**. We would then jump to the end of the program using **b**. If we jumped to **next**, then we subtracted 2 from **r1** then stored the value into **r1**.

After writing our program using **nano**, we then assembled and linked it using **as** and **ld** respectively. We then used **gdb** to begin debugging our program. We set a breakpoint using the **b** command, then used **stepi** until we were at the line we were interested in. In this case, we wanted to be at the last line of this program to see if the value stored in **x** was decreased by 1 or 2. We used **info registers** to display the values of the registers at this point, then used **x/1xw** to display the value stored in **x** as a hexadecimal number. Below is the result:



From this menu, we can see that the value stored in **x** is 0 and that the **Z flag** is set to 1. This means that our program worked as intended, as the initial value of **x** was 1, which is <=3. As such, we only subtracted 1 from **x**.

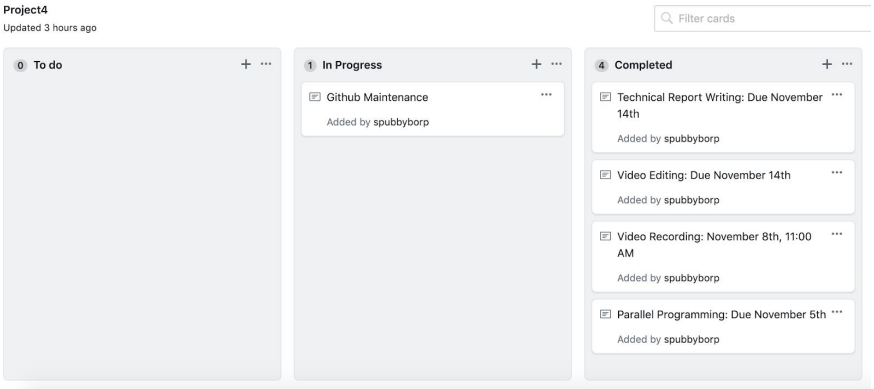
4. Appendix

4.1. Github

4.1.1. Link:

https://github.com/Team-Chabu/CSC3210-CHABU

4.1.2. Screenshot:



4.2. Slack

https://join.slack.com/t/csc-3210-teamchabu/shared_invite/enQtNzc0ODEwMjkyNjU2LTU4MGQ2MzcyMDg5YTJhMjc4ZmZjZWU4OTgzNjk4NzI1M2ZjMjY5YjBiMjYyMTgyOGVmNGQzMjFhMjYzN2YxOWM

4.3. Youtube (Channel)

https://www.youtube.com/channel/UCp-0kVA08dU4F79t6slHtLQ

4.4. Youtube (Video)

https://youtu.be/0uHnhZ81oCM