Developing Soft and Parallel Programming Skills Using Project-Based Learning

Fall-2018

Team Nine30

Noah-Raine De Joya, Angelica Amis, Frankie Sanchez, Sai Pavan Nuthalapati, Phillip King

Planning and Scheduling:

Assignee Name	Email	Task	Duration (Hours)	Dependency	Due Date	Notes
Frankie Sanchez	fsanchez4@student.gsu.edu	In charge of the programming portion of Task 3 Part B: Parallel Programming Basics	4 hours	Raspberry PI hardware & the tutorials contained in the assignment	10/26/18	Created detailed lab report to describe observations of parallel programming
Phillip King (Coordinator)	pking17@student.gsu.edu	Assist with programming Task 3 – Part B: Parallel Programming Basics; Task 3 – Record and upload YouTube video to Team Nine30 channel	hours	Raspberry PI hardware; YouTube; Team member involvement for the video recording session	10/26/18	Recorded videos on 10/24/18 after class
Angelica Amis	aamis1@student.gsu.edu	Task 5 – Report all tasks done by group members through written communication	1 hours	Information needed for report append	10/26/18	Touched up by answering missing questions and finishing chart
Sai Pavan Nuthalapati	snuthalapati@student.gsu.edu	Create & maintain GitHub as outlined in Task 2: Collaboration; Answer Task 3 questions Part A: Foundation	2 hours	GitHub	10/26/18 (GitHub); 10/26/18 (Report)	Periodically maintain & update the work progress on GitHub
Noah De Joya	ndejoya1@student.gsu.edu	Work Breakdown Rubric of Task 1 – Planning and Scheduling; Assist team with Task 3 – Foundation Questions	2 hours	None	10/26/18	Created detailed chart breaking down the duties of each group member

Parallel Architecture and Programming Skills

Foundation:

1. Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.
    - Task: A task is typically a program or program-like set of instructions that is executed by a processor.
    - Pipelining: Breaking a task into steps performed by different processor units, with inputs streaming through.
    - Shared Memory: A computer architecture where all processors have direct (usually bus based) access to common physical memory.
    - Communications: The event of data exchange through a shared memory bus or over a network, to name a few, for parallel tasks.
    - Synchronization: The coordination of parallel tasks in real time, very often associated with communications.
2. Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them
    - Single Instruction, Single Data (SISD): A serial computer which only has single instruction stream and single data stream during any one clock cycle.
    - Single Instruction, Multiple Data (SIMD): one of parallel computer which execute the same instruction at any given clock cycle and processing unit is capable of operating on a different data element.
    - Multiple Instruction, Single Data (MISD): Each processing unit operates on the data independently via separate instruction streams, a single data stream is fed into multiple processing units.
    - Multiple Instruction, Multiple Data: Every processor may be executing a different instruction stream, every processor may be working with a different data stream
3. What are the Parallel Programming Models?
   Ans: Shared memory (without threads), threads, distributed memory/ message passing,

   data parallel, hybrid, Single Program Multiple Data (SPMD), Multiple Program

   Multiple Data (MPMD)

4. List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?
    - Uniform Memory Access (UMA): Identical processors with equal access and access times to memory. All the processors are aware of any updates made to any one processor.
    - Non-Uniform Memory Access (NUMA): One Symmetric Multiprocessor (SMP) can directly access memory of another SMP and not all processors have equal access time to all memories.
5. Compare Shared Memory Model with Threads Model?
   Ans: In shared memory, processes and tasks have an equal access to the memory for reading and writing. In a thread model a single process can have multiple threads running at the same time.
6. What is Parallel Programming?

Ans: Parallel programming increase the available computation power for faster application processing or task resolution. It is the execution of multiple processes or threads simultaneously through either a single or multiple processor.

7. What is system on chip(SoC)? Does Raspberry PI use system on SoC
   - A system on chip(SoC) is a microchip which is capable to handle all the tasks run on a CPU without any other external chips supporting it. It contains GPU, memory, USB controller, power management circuits, and wireless radios.
   - Raspberry PI uses system on chip (SoC)
8. Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.
   - The size of CPU is a bit smaller than SOC but SOC accommodates all the required components without any additional chips.
   - Due to its very high level of integration and much shorter wiring, an SoC also uses considerably less power.
   - In today's world and in future we expect electronics to be sleeker, lightweight, small, and with long lasting battery power. SoCs takes less space while providing good computing power and provides more space for batteries.
   - In today's market we have many manufactures making phones, tablets and other electronics with pricing ranging from really cheap to expensive. SoCs requires only one chip which costs less than manufacturing multiple chips this makes product manufacture cheap in return making products cheaper.

Parallel Programming Basics:

Program 1: parallelLoopEqualChunks.c

$ nano parallelLoopEqualChunks.c

```
  GNU nano 2.7.4                                                    File:

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

int main(int argc, char** argv) {
const int REPS = 16;

printf("\n");
if(argc > 1) {
        omp_set_num_threads(atoi(argv[1]));
}

#pragma omp parallel for
for(int i = 0; i < REPS; i++) {
        int id = omp_get_thread_num();
        printf("Thread %d performed iteration %d\n", id, i);
}

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

Here we typed out the C file in the terminal window using nano. To compile the program and rename it *pLoop*, we performed the following:

$ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp

```
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ls
Desktop  Documents  Downloads  MagPi  Music  parallelLoopEqualChunks.c  Pictures  pLoop  Public  python_games
```

To make sure our file was created, we utilized the 'ls' utility. Then, we ran the program with 4 threads by using 4 as the argument. Since the Raspberry Pi has 4 cores, it was natural to try 4 threads.

$ ./pLoop 4

```
pi@raspberrypi:~ $ ./pLoop 4

Thread 2 performed iteration 8
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 3 performed iteration 14
Thread 3 performed iteration 15
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 0 performed iteration 3
```

It is interesting to note that the order of the iterations as displayed on the terminal follows no specific pattern, with the exception of threads undertaking consecutive iterations. One would think that the iterations would move in order, or the order of when the thread executes, but this is not the case. The more important take away is that thread 0 manages the first 4 iterations (0-3), thread 1 manages the second 4 iterations (4-7), thread 2 manages the third 4 iterations (8-11), and thread 3 manages the last 4 iterations (12-15). Below we will run the same program using arguments 3, 2, 1 and no argument.

$ ./pLoop 3

```
pi@raspberrypi:~ $ ./pLoop 3

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 0 performed iteration 4
Thread 0 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 1 performed iteration 8
Thread 1 performed iteration 9
Thread 1 performed iteration 10
Thread 2 performed iteration 11
Thread 2 performed iteration 12
Thread 2 performed iteration 13
Thread 2 performed iteration 14
Thread 2 performed iteration 15
```

With 3 as the argument, thread 0 performed 6 iterations (0-5), thread 1 performed 5 iterations (6-10), and thread 2 performed 5 iterations (11-15). The interesting takeaway is that thread 0 had one additional iteration in comparison to the threads 1 and 2.

$ ./pLoop2

```
pi@raspberrypi:~ $ ./pLoop 2

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 0 performed iteration 4
Thread 0 performed iteration 5
Thread 0 performed iteration 6
Thread 0 performed iteration 7
Thread 1 performed iteration 8
Thread 1 performed iteration 9
Thread 1 performed iteration 10
Thread 1 performed iteration 11
Thread 1 performed iteration 12
Thread 1 performed iteration 13
Thread 1 performed iteration 14
Thread 1 performed iteration 15
```

With 2 as the argument, thread 0 performed 8 iterations (0-7)and thread 1 performed 8 iterations (8-15). The total number of iterations was evenly divided among the 2 threads.

$ ./pLoop 1

```
pi@raspberrypi:~ $ ./pLoop 1

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 0 performed iteration 4
Thread 0 performed iteration 5
Thread 0 performed iteration 6
Thread 0 performed iteration 7
Thread 0 performed iteration 8
Thread 0 performed iteration 9
Thread 0 performed iteration 10
Thread 0 performed iteration 11
Thread 0 performed iteration 12
Thread 0 performed iteration 13
Thread 0 performed iteration 14
Thread 0 performed iteration 15
```

With 1 as the argument, thread 0 performed all iterations. This is not surprising.

$ ./pLoop

```
pi@raspberrypi:~ $ ./pLoop

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 2 performed iteration 8
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 3 performed iteration 14
Thread 3 performed iteration 15
```

With no argument, it’s surprising to see that the threads were set according to the number of available cores (4 cores = 4 threads). Again, the output is like that of the output with 4 as the argument.

It appears that a modulo operator is used to calculate the remainder when the number of iterations is not evenly divisible by the number of threads. One iteration is added on to each lower thread until the remaining value (remainder = remaining iterations) is depleted. For example, in the *pLoop* program with 3 as the argument, thread 0 took on the additional iteration. We know we have 16 iterations, and we have 3 as the argument. 16 mod 3 = 1. With 1 as the remainder, this was added onto the lowest thread, thread 0, giving it 6 iterations; while the other threads undertook 5 iterations.

Program 2: parallelLoopChucksof1.c

$ nano parallelLoopChunksof1.c

```
  GNU nano 2.7.4                                                    File:

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

int main(int argc, char** argv) {
const int REPS = 16;

printf("\n");
if(argc > 1) {
        omp_set_num_threads(atoi(argv[1]));
}

#pragma omp parallel for schedule(static, 1)
for(int i = 0; i < REPS; i++) {
        int id = omp_get_thread_num();
        printf("Thread %d performed iteration %d\n", id, i);
}

/*
printf("\n---\n\n");

#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
for(int i = id; i < REPS; i+=numThreads) {
        printf("Thread %d performed iteration %d \n",
                a.      id, i);
}
}
*/

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

Here we typed out the C file in the terminal window using nano. To compile the program and rename it *pLoop2*, we performed the following:

$ gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp

```
pi@raspberrypi:~ $ gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ ls
Desktop  Documents  Downloads  MagPi  Music  parallelLoopChunksOf1.c  parallelLoopEqualChunks.c  Pictures  pLoop  pLoop2
pi@raspberrypi:~ $
```

To make sure our file was created, we utilized the ‘ls’ utility. Then, we ran the program with 4 threads by using 4 as the argument. Since the Raspberry Pi has 4 cores, it was natural to try 4 threads.

$ ./pLoop2 4

```
pi@raspberrypi:~ $ ./pLoop2 4

Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 2 performed iteration 10
Thread 2 performed iteration 14
```

After running the program with an argument of 4, we can see that each thread computes an equal amount of work, just not consecutive iterations. In this case, each thread completes 1 (chunk size) iteration and moves on to the next thread/iteration combination (i.e. 0 thread performs iteration 0, 1 thread performs iteration 1, thread 2 performs iteration 2, and so on until the 4 threads complete the first 4 iterations; then repeats until all iterations are complete). This is surprising in the sense that declaring a static assignment can dictate the amount of work and order by which threads can compute work. Below we will replace 4 with other values for the number of threads.

./pLoop2 3

```
pi@raspberrypi:~ $ ./pLoop2 3

Thread 1 performed iteration 1
Thread 1 performed iteration 4
Thread 1 performed iteration 7
Thread 1 performed iteration 10
Thread 1 performed iteration 13
Thread 0 performed iteration 0
Thread 0 performed iteration 3
Thread 0 performed iteration 6
Thread 0 performed iteration 9
Thread 0 performed iteration 12
Thread 0 performed iteration 15
Thread 2 performed iteration 2
Thread 2 performed iteration 5
Thread 2 performed iteration 8
Thread 2 performed iteration 11
Thread 2 performed iteration 14
```

With 3 as the argument, we can see the number of threads is set to 3 threads (threads 0-2). With static being set and assigned a value of 1 (chunk size), we schedule each thread to do one iteration of the loop in a regular pattern. Also, we notice that the remainder principle still applies. Thread 0 has 6 iterations (0, 3, 9, 12, and 15), thread 1 has 5 iterations (1, 4, 7, 10, and 13), and thread 2 has 5 iterations (2, 5, 8, 11, and 14). Since 16 % 3 = 1, the remaining value 1 was added to thread 0—the lowest thread—giving it an extra iteration.

./pLoop2 2

```
pi@raspberrypi:~ $ ./pLoop2 2

Thread 0 performed iteration 0
Thread 0 performed iteration 2
Thread 1 performed iteration 1
Thread 1 performed iteration 3
Thread 1 performed iteration 5
Thread 1 performed iteration 7
Thread 1 performed iteration 9
Thread 1 performed iteration 11
Thread 1 performed iteration 13
Thread 1 performed iteration 15
Thread 0 performed iteration 4
Thread 0 performed iteration 6
Thread 0 performed iteration 8
Thread 0 performed iteration 10
Thread 0 performed iteration 12
Thread 0 performed iteration 14
```

With 2 as the argument, we see the same process with an equal number of iterations between the two threads 0 and 1.

./pLoop2 1

```
pi@raspberrypi:~ $ ./pLoop2 1

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 0 performed iteration 4
Thread 0 performed iteration 5
Thread 0 performed iteration 6
Thread 0 performed iteration 7
Thread 0 performed iteration 8
Thread 0 performed iteration 9
Thread 0 performed iteration 10
Thread 0 performed iteration 11
Thread 0 performed iteration 12
Thread 0 performed iteration 13
Thread 0 performed iteration 14
Thread 0 performed iteration 15
```

Thread 0 handles all the iterations. Although static and chunk size are set, there is only 1 thread which doesn't meet the argument size requirement for the *if-statement*.

./pLoop2

```
pi@raspberrypi:~ $ ./pLoop2

Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 2 performed iteration 10
Thread 2 performed iteration 14
Thread 3 performed iteration 11
Thread 3 performed iteration 15
```

The output of the *pLoop2* program without arguments is similar to the output when the argument is set to 4 (4 cores = 4 threads). This result is expected.

We can also assign the threads dynamically by changing *static* to *dynamic*. Under this assignment, when a thread completes it gets the next iteration or chuck still needed to be completed. For example:

#pragma omp parallel for schedule (**dynamic**, 1)

Program 3: reduction.c

$ nano reduction.c

```
  GNU nano 2.7.4

// reduction.c

#include <stdio.h>      // printf()
#include <omp.h>        // OpenMP
#include <stdlib.h>     // rand()

void initialize(int* a, int n);
int sequentialSum(int* a, int n);
int parallelSum(int* a, int n);

#define SIZE 1000000

int main(int argc, char** argv) {
int array[SIZE];

if(argc > 1) {
        omp_set_num_threads(atoi(argv[1]));
}

initialize(array, SIZE);
printf("\nSequential sum: \t%d\nParallel sum: \t\t%d\n\n",
sequentialSum(array, SIZE),
parallelSum(array, SIZE));

return 0;
}

// fill array with random values
void initialize(int* a, int n) {
        int i;
        for(i = 0; i < n; i++) {
                a[i] = rand() % 1000;
        }
}

// 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;
}
```

Here we typed out the C file in the terminal window using nano. To compile the program and rename it *reduction*, we performed the following:

$ gcc reduction.c -o reduction -fopenmp

After compiling we ran the program with arguments 4, 3, and no arguments. This is shown below.

./reduction 4

```
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:           499562283

pi@raspberrypi:~ $
```

/.reduction 3

```
pi@raspberrypi:~ $ ./reduction 3

Sequential sum:         499562283
Parallel sum:           499562283
```

/.reduction

```
pi@raspberrypi:~ $ ./reduction

Sequential sum:         499562283
Parallel sum:           499562283
```

In this program, an array of randomly assigned integers is a set of data shared between both sequentialSum() and parallelSum() functions. The functions are identical while the #pragma line is commented out. Thus, we can expect the for loops to sum up all the values in the array. This results in the same output for each function. The correct output is produced by both sequentialSum() and parallelSum() in this example.

Now we will remove the // in front of #pragma in line 39, re-compile, and re-run the program. This will uncomment the first part of the #pragma line allowing the program to run in parallel. The output after running the program with arguments 4, 3, 2, 1, and no arguments are below:

$ ./reduction *n* ; (where n = 4, 3, 2, 1, and " ")

```
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:           161847880
```

```
pi@raspberrypi:~ $ ./reduction 3

Sequential sum:         499562283
Parallel sum:           216737922
```

```
pi@raspberrypi:~ $ ./reduction 2

Sequential sum:         499562283
Parallel sum:           300145911
```

```
pi@raspberrypi:~ $ ./reduction 1

Sequential sum:         499562283
Parallel sum:           499562283
```

```
pi@raspberrypi:~ $ ./reduction

Sequential sum:         499562283
Parallel sum:           158747670
```

Here we immediately notice the sequentialSum() function's results do not match the parallelSum() function's results, apart from when the argument is set to 1. This makes sense because to set the number of threads, the argument must be a value greater than 1. Here parallelSum() does not produce the correct output. This is interesting as we are allowing parallel computation; however, without the reduction clause the threads cannot implicitly communicate to keep the overall sum updated as each work on a portion of the array.

Next we removed the second // in line 39, re-compiled and re-ran the program using arguments of 4, 3, 2, 1, and no argument. The output is shown below.

$ ./reduction *n* (where n = 4, 3, 2, 1, " ")

```
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:           499562283

pi@raspberrypi:~ $ ./reduction 3

Sequential sum:         499562283
Parallel sum:           499562283

pi@raspberrypi:~ $ ./reduction 2

Sequential sum:         499562283
Parallel sum:           499562283

pi@raspberrypi:~ $ ./reduction 1

Sequential sum:         499562283
Parallel sum:           499562283

pi@raspberrypi:~ $ ./reduction

Sequential sum:         499562283
Parallel sum:           499562283
```

The resulting sum of both functions now equals the correct output. This is what should happen as the reduction clause is now available to be read and executed. The reduction clause, reduction(+:sum), allows all the values to be summed together by using the OpenMP parallel for pragma. The plus sign in the pragma reduction clause signals the variable sum is being computed by added values together in the loop. The threads are now communicating to keep the overall sum updated as each of them works on the portion of the array.

We believe the parallel for pragma did not produce the correct result without the reduction clause because the threads have no way of communicating to track the sums of each iteration. This communication is paramount to return a sum of all computations performed by each thread. The accumulator sum needs to be private as each thread completes its work. Then, when each thread is finished, the final sum of their individual sums is computed. The variable sum is dependent on what the other threads are doing.

A full snapshot of program 3 is below.

```
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:           499562283

pi@raspberrypi:~ $ ./reduction 2

Sequential sum:         499562283
Parallel sum:           499562283

pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:           162496098

pi@raspberrypi:~ $ ./reduction 2

Sequential sum:         499562283
Parallel sum:           294772513

pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:           499562283

pi@raspberrypi:~ $ ./reduction 2

Sequential sum:         499562283
Parallel sum:           499562283
```

Appendix:

Links –

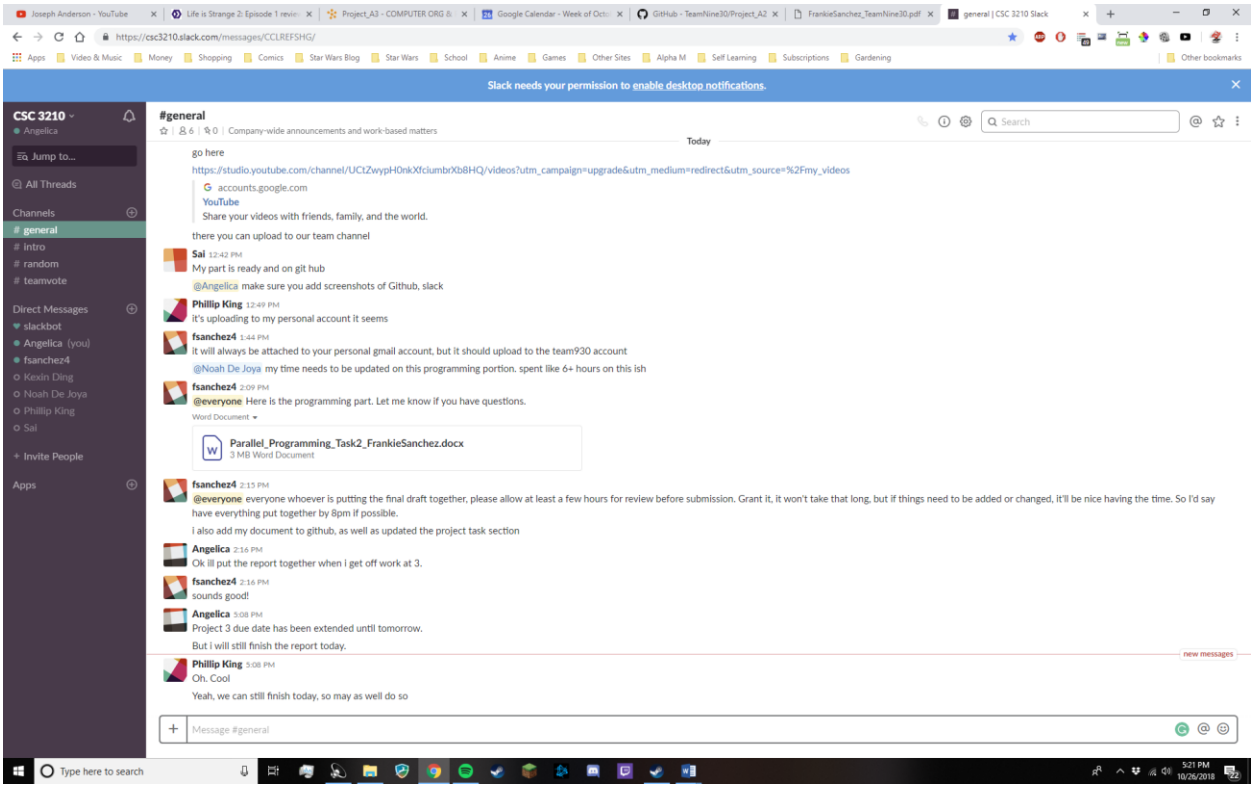
Slack: https://csc3210.slack.com/messages/CCLREFSHG/

GitHub: https://github.com/TeamNine30

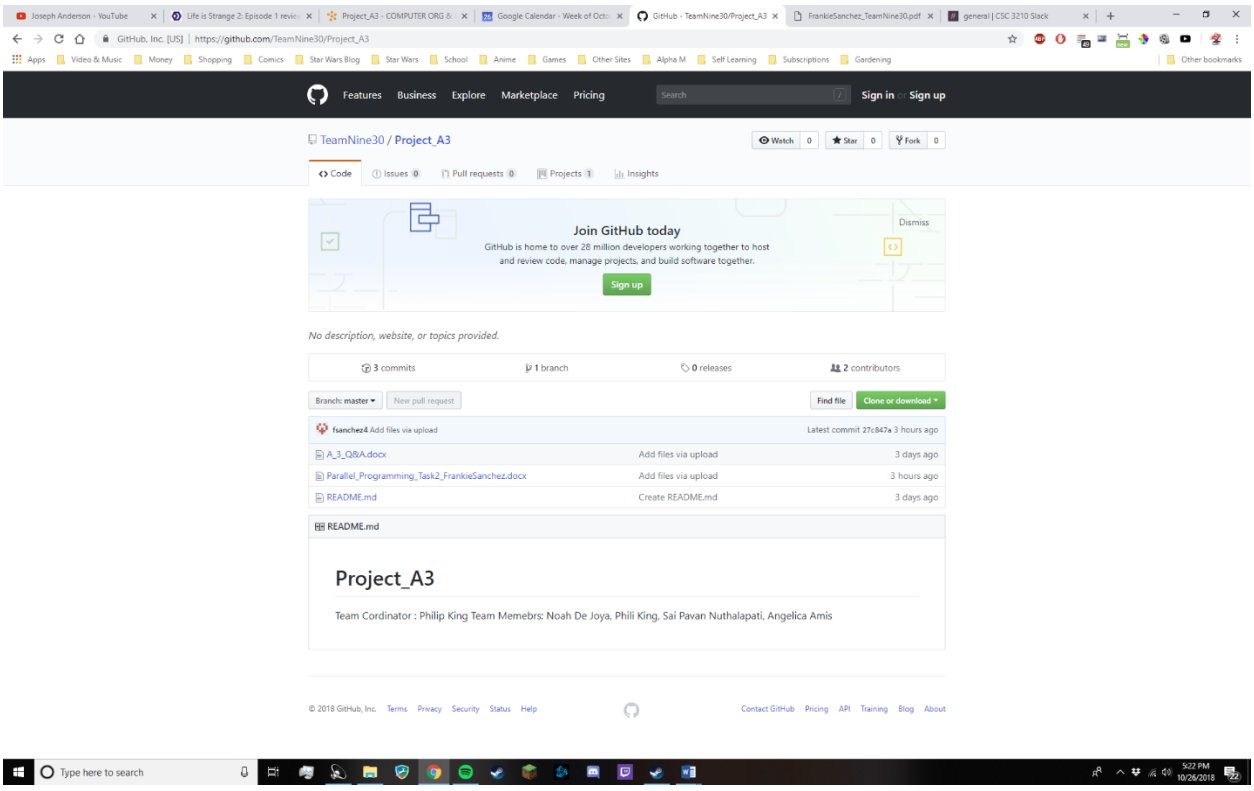
YouTube Channel Link: https://www.youtube.com/channel/UCtZwypH0nkXfciumbrXb8HQ

Screen Shots –

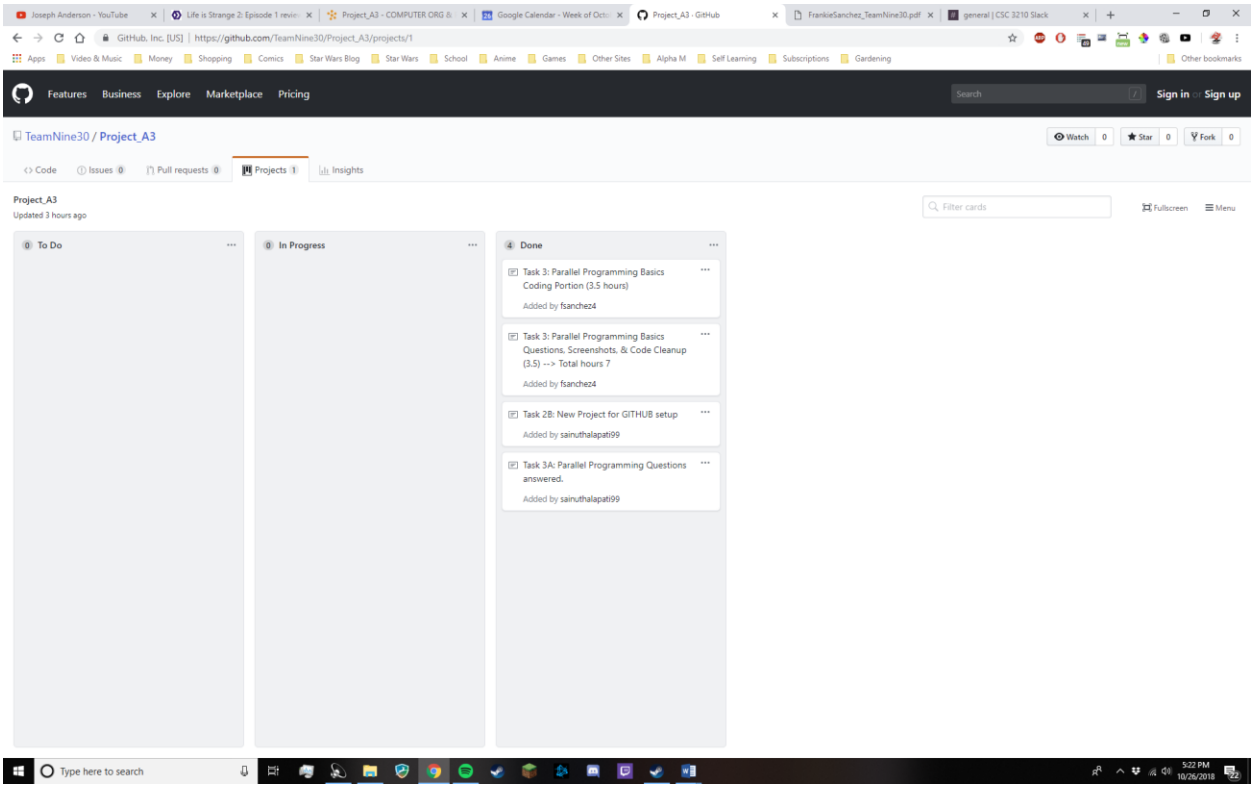
Slack:



GitHub:


Features Business Explore Marketplace Pricing
Sign in or Sign up
TeamNine30 / Project_A3
Watch 0 Star 0 Fork 0
Code Issues 0 Pull requests 0 Projects 1 Insights
Join GitHub today
GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.
Sign up
Dismiss
No description, website, or topics provided.
3 commits 1 branch 0 releases 2 contributors
Branch: master New pull request Find file Clone or download
fsanchez4 Add files via upload Latest commit 27c847a 3 hours ago
A_3_Q&A.docx Add files via upload 3 days ago
Parallel_Programming_Task2_FrankieSanchez.docx Add files via upload 3 hours ago
README.md Create README.md 3 days ago
README.md
Project_A3
Team Cordinator : Philip King Team Memebrs: Noah De Joya, Phili King, Sai Pavan Nuthalapati, Angelica Amis
© 2018 GitHub, Inc. Terms Privacy Security Status Help
Contact GitHub Pricing API Training Blog About

Type here to search
5:22 PM
10/26/2018


Features Business Explore Marketplace Pricing
Sign in or Sign up
TeamNine30 / Project_A3
Watch 0 Star 0 Fork 0
Code Issues 0 Pull requests 0 Projects 1 Insights
Project_A3
Updated 3 hours ago
Filter cards
Fullscreen Menu
0 To Do
0 In Progress
4 Done
Task 3: Parallel Programming Basics Coding Portion (3.5 hours)
Added by fsanchez4
Task 3: Parallel Programming Basics Questions, Screenshots, & Code Cleanup (3.5) --> Total hours 7
Added by fsanchez4
Task 2B: New Project for GITHUB setup
Added by sainuthalapati99
Task 3A: Parallel Programming Questions answered.
Added by sainuthalapati99

Type here to search
5:22 PM
10/26/2018