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	Email	Task	Duration(hours)	Dependency	Due Date	Note
Manuel Lipo (Coordinator)	Mlipo1@student.gsu.edu	Create chart, Programming, Team coordinator, Turning Final report in.	3hrs	Table must be done before Monday. Assign everyone's roles and finish work	11/15/18	
Jimmy Homeis	jpetithomeis@student.gsu.edu	Video editing and filming. Programming assignment on Raspberry pi. Answering Questions	3hrs	Videos must be edited, Uploaded and finished before next Monday, Answer questions	11/15/18	Includes Programming skills
Brain Cabau	Bcabau1@student.gsu.edu	GitHub, Raspberry pi Programming assignment. Written Lab report, Answering questions.	3hrs	Pi assignment must be finished. Written lab report needs to be started and finished by 11/14	11/15/18	Includes programming skills and basics
Riyazh Dholakia	Rdholakia1@student.gsu.edu	Raspberry pi programming, working on GitHub and uploading files. Answering Questions and helping with lab report.	3hrs	Work on raspberry pi and start working on lab questions and video assignment	11/15/18	Includes programming skills and basics and GitHub
Anli Ji	Aji1@student.gsu.edu	Writing and formatting the lab report, helping with GitHub and answering questions.	3hrs	Lab report needs to be finished up and uploaded by 11/14. Work on Pi assignment	11/15/18	Document the lab report along with everyone else. Questions.

Task 2 Parallel Architecture and Programming Skills:

a) Foundation

What is race condition?

Race condition or race hazard is when the behavior is depended on the other uncontrollable events or even sequences. This behavior can happen with electronics, software, or even other systems such as logic circuits, multithreaded, or distributed software programs. This idea can be used for almost anything. Even our daily life, but race condition is more based towards the meaning of hardware and software. These uncontrollable events can cause a lot of problems and bugs. An example would be asynchronization, but a larger scale where would not be easily to manipulate the bug or problem. Another example is with our multi-threads when running our C code in raspberry pi terminal.

Why race condition is difficult to reproduce and debug?

For software, the condition is difficult to reproduce and debug because the conditions are nondeterministic. This means even with the same input the output can be different. In our cases, it is the threads changing our output since they "finish" at different times. This was throwing output way out of order, and not sequentially the way we are expecting it to be. This is just one of the many cases where race condition is being difficult. A good developer can figure this situation and track down a bug and manipulate it the way he/she wants to output to be.

How can it be fixed? Provide an example from your Project_A3(see spmd2.c)

Race conditions are hard to fix because diagnosing the problem is difficult as errors are not concurrent. The only way to fix errors that don't always show is to understand the code and how the code is utilizing the threads. Races are usually run into when multiple threads are attempting to access the same data and runs into an error of synchronization due to the order of sequence not always being the same. A proper solution is to understand the code and include proper locks into which threads are being used.

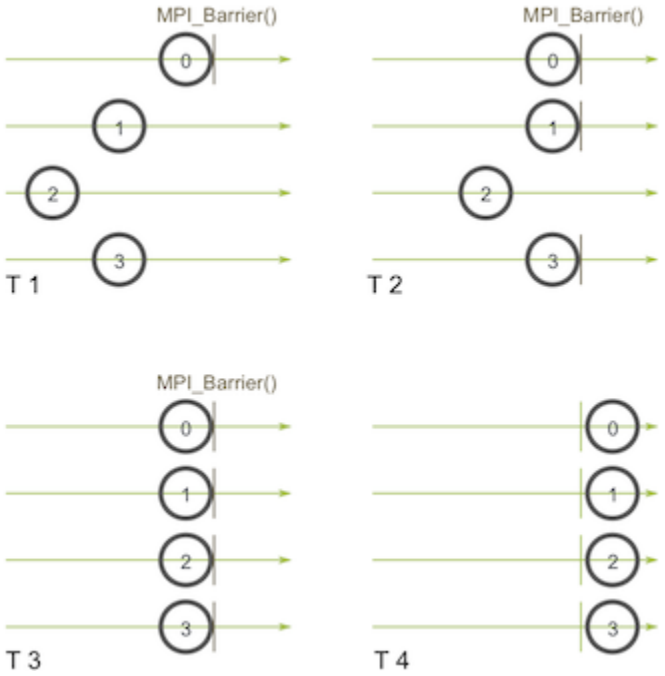
Summaries the Parallel Programming Patterns section in the "Introduction to Parallel Computing_3.pdf" in your own words (one paragraph, no more than 150 words).

Parallel Programming Patterns as the name stated is a set of instructions that involves the path of many patterns. This pattern are useful ways of writing code for parallel programming that help programmers to have a better understanding of coding and to be useful for other developers. Parallel Programming involves different patterns.

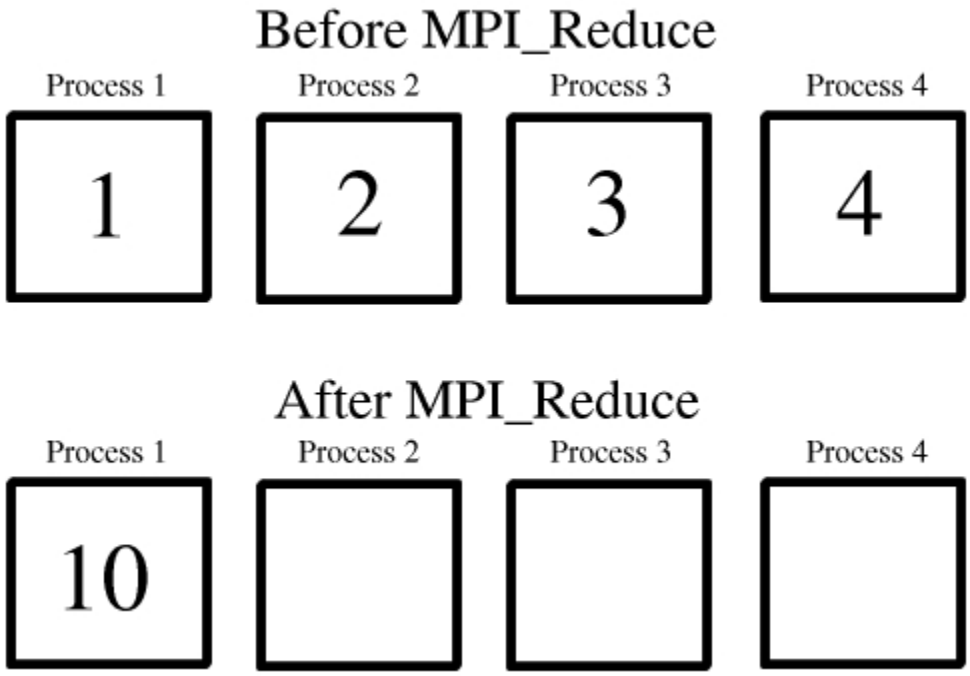
In the section "Categorizing Patterns" in the "Introduction to Parallel Computing_3.pdf" compare the following:
- Collective synchronization (barrier) with Collective communication (reduction)

Collective synchronization and Collective communication are both under the category of coordination pattern. In Collective synchronization, every single collective call that has been

made is synchronized. As for an example, the barrier in collective synchronization, not all of the processes in the communicator can pass through the barrier until all of them call the function.



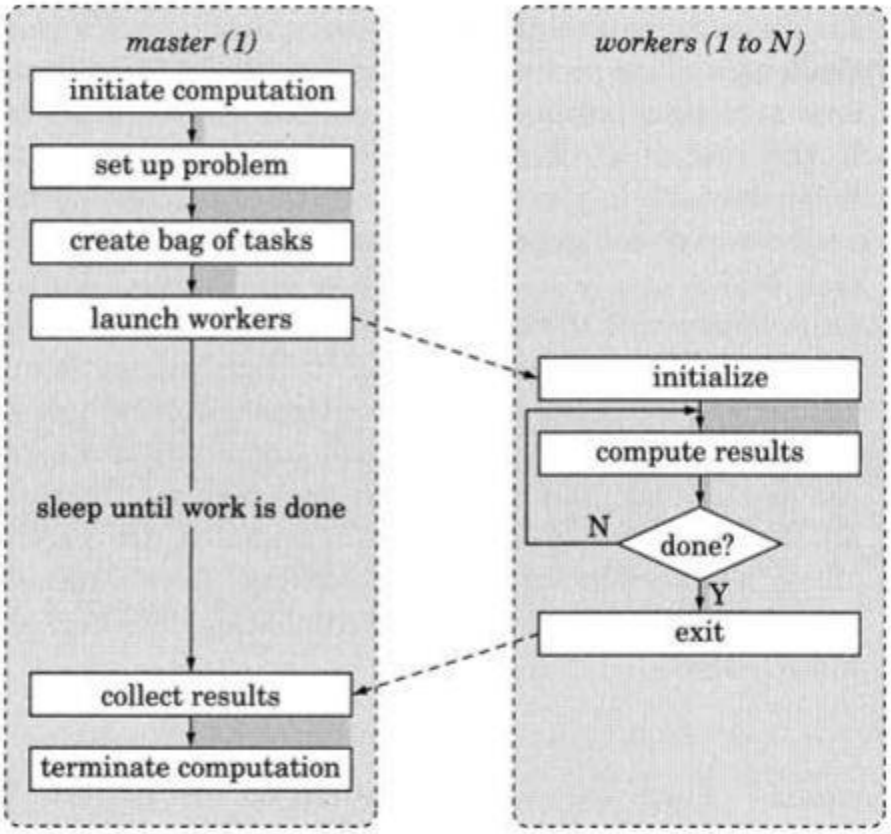
Collective communication is defined as communication that involves the participation of a group of processes. The global reduction including sum, max, min, or user-defined functions will return a result to every member in the group and a variation where the result will only be returned to one single member.



-Master-worker with fork-join

Both master-worker and fork-join are program structures that we used in implementation strategies. In master-worker, there will be a “master” (main process) and one or more “workers” (individual process). The main master will be divided into several different tasks which will be dispatched to the different worker process.

Fork-Join is similar to master-worker but in a way that the main process will continue to execute while the forks are processing. In another word, each task from the main manager will be forked and joined back to the parent task after execution. It is more likely a parallel processing instead of monodirectional processing.



b) Parallel Programming Basics

The first program is meant to compute an approximation of the integral of sin(x) with bounds from 0 to pi. The program computes areas of 1048576 trapezoids in order to approximate the value of the integral to a certain precision. Then the areas are summed. This task can easily be solved using parallel programming, by assigning a number of subdivisions to a number of threads, and consequently speeding up the process of solving for the integral.

Below, we see two code snippets with key differences on line 37, the line responsible for parallelizing the loop, and defining the scope of relevant variables. While both programs have a result of 2 for the integral, the code on the right has the correct implementation using the reduction clause. The accumulator variable should be private to each thread until the end where the final summation of the integral would be computed.

```
#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);

}
```

```
#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 next program shows a Barrier synchronization, which has to do with the timing of threads running in parallel. This causes threads to stop from moving on, until all of the threads have reached the same point in the computation. The below code and output shows synchronization using barrier clearly. When the barrier line is commented, the output shows each thread computing both the before and after portion of the program sequentially.

```
#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 barries.\n",id,numThreads);
                }

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

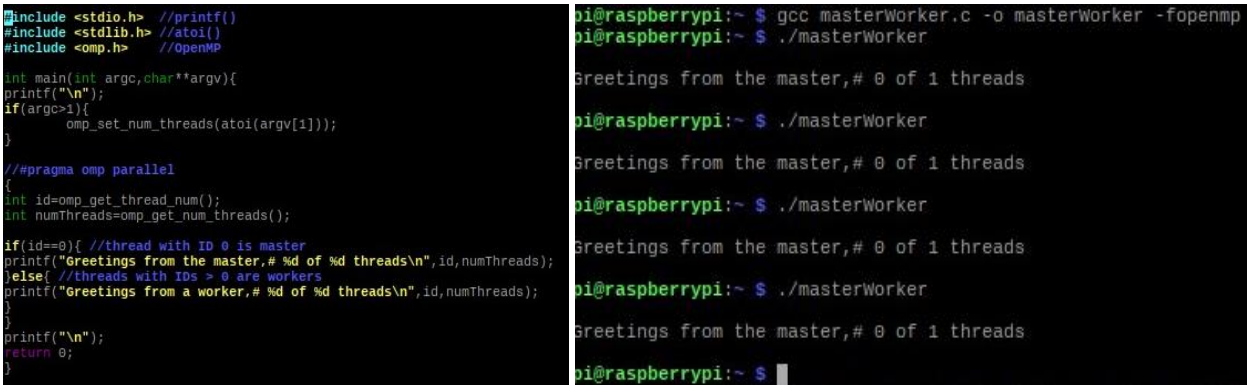
```
pi@raspberrypi:~ $ ./barrier

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

pi@raspberrypi:~ $ nano barrier.c
pi@raspberrypi:~ $
```

When the barrier line is uncommented, the output shows that all threads complete the before portion before continuing to compute the after portion of the code. So barrier halts the progress of the program until all threads have done the first part.

```
#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 barries.\n",id,numThreads);
                }

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

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

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

pi@raspberrypi:~ $
```

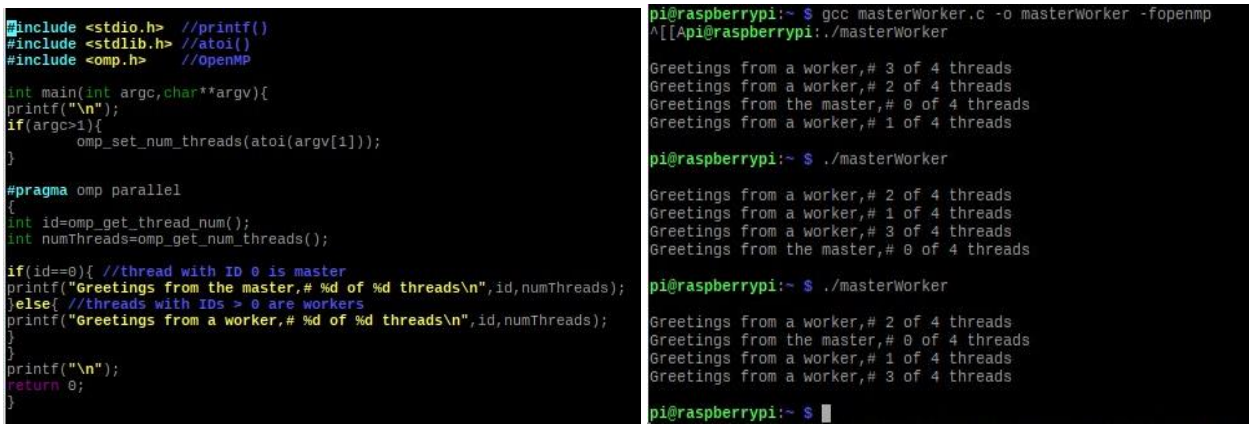
The final program utilizes the Master-Worker Implementation. This implementation splits computation between a master thread, followed by a number of worker threads. In this example while the pragma line is commented, the output only shows the computation of the thread that was the masters. The worker threads don't run because their threads must be greater than 0, since the master's thread is thread 0. The program is not running in parallel due to the line being commented, and as a result, the output only shows thread 0's execution.



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

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){ //thread with ID 0 is master
printf("Greetings from the master,# %d of %d threads\n",id,numThreads);
}else{ //threads with IDs > 0 are workers
printf("Greetings from a worker,# %d of %d threads\n",id,numThreads);
}
}
printf("\n");
return 0;
}
```

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

Greetings from the master,# 0 of 1 threads

pi@raspberrypi:~ $ ./masterWorker

Greetings from the master,# 0 of 1 threads

pi@raspberrypi:~ $ ./masterWorker

Greetings from the master,# 0 of 1 threads

pi@raspberrypi:~ $ ./masterWorker

Greetings from the master,# 0 of 1 threads

pi@raspberrypi:~ $
```

If the pragma line is uncommented, the output changes completely, because the program is running on multiple threads. The master thread is 0, and the worker threads go on from there, and the output shows the workers as well as the master. This is a simple example, but it seems like this implementation might be useful if there is one main task for the master to complete, and subsidiary tasks for the workers to complete, maybe based on some hierarchy of the priorities within the code.



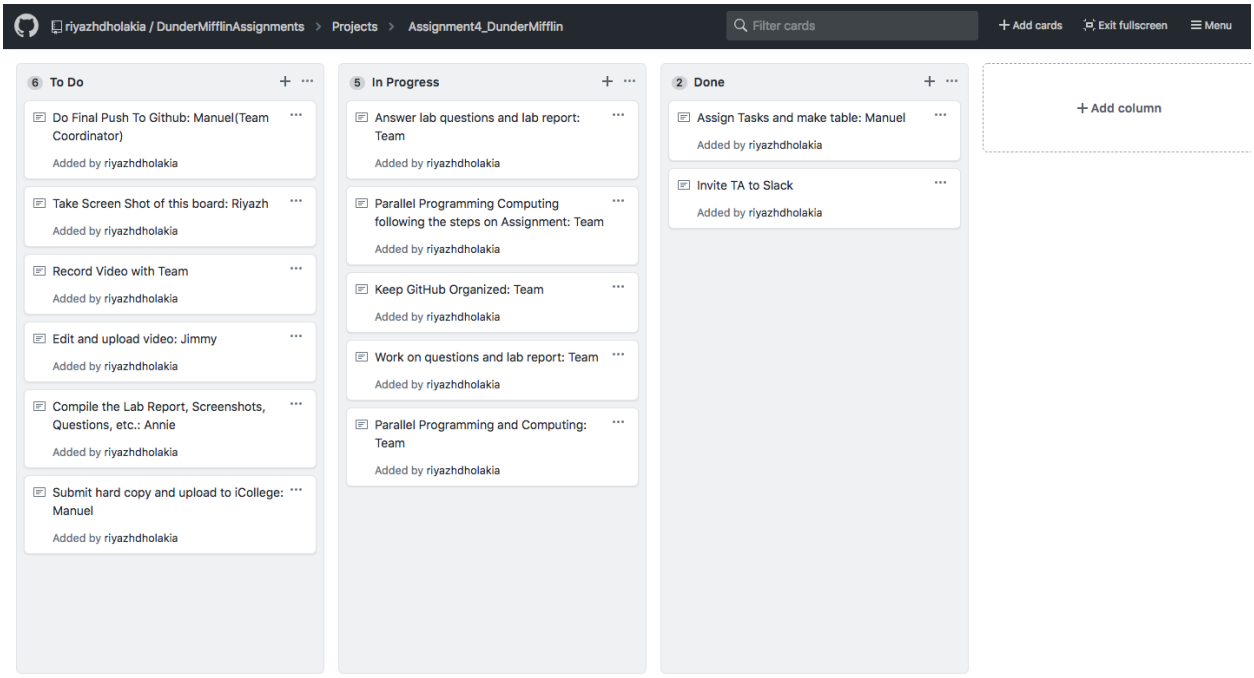
```c
#include <stdio.h>  //printf()
#include <stdlib.h> //atoi()
#include <omp.h>    //OpenMP

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){ //thread with ID 0 is master
printf("Greetings from the master,# %d of %d threads\n",id,numThreads);
}else{ //threads with IDs > 0 are workers
printf("Greetings from a worker,# %d of %d threads\n",id,numThreads);
}
}
printf("\n");
return 0;
}
```

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

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

pi@raspberrypi:~ $ ./masterWorker

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

pi@raspberrypi:~ $ ./masterWorker

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

pi@raspberrypi:~ $
```

Task 3 Appendix:

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

Github Project Screen Shot:



YouTube link:

Video Link:

https://youtu.be/CIzoJ1QHVBw

Channel Link:

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