Developing Soft and Parallel Programming Skills Using Project-Based Learning

Fall 2018, Group 5

Austin Nocero, Rickey Clark, Paige Park, Nicholas Economou, Zhiyi Dong

Catalog----

Planning And Scheduling

Parallel Programming Skills

Appendix: Content Links

Planning and Scheduling:

Assignee Name	**Email**	**Task**	**Duration (Hours)**	**Dependency**	**Due Date**	**Note**
Nicholas Economou (Coordinator)	neconomou1@student.gsu.edu	3B Parallel programming basics & YouTube	2 hours	Video clips, Parallel Progra-mming	11/15/18	Must be ready 24 hours before the due date
Paige Park	ppark11@student.gsu.edu	Task5 Report & GitHub	2 hours	Lab reports	11/16/18	Must be ready on the first day of group work
Zhiyi Dong	zdong4@student.gsu.edu	3A parallel programming skill questions	3 hours	Introduction to Parallel Computing	11/15/18	Must be ready 5 hours before deadline
Austin Nocero	anocero1@student.gsu.edu	3A parallel programming skill questions	3 hours	Introduction to Parallel Computing	11/15/18	Must be ready 5 hours before deadline
Rickey Clark	rclark39@student.gsu.edu	3B Parallel programming basics	3 hours	Parallel Progra-mming	11/15/18	Must be ready 5 hours before deadline

Parallel Programming Skills:

Foundation: Use the reading material to answer the questions (in your own words)

Question 1) What is race condition?

Race condition is where the timing of events is out of sync and does not match up. This then effects the order that the program intended to have the events go in. For example, if two operations are done at the same time then one of them must go before the other and the sequence could be incorrect.

Question 2) Why race condition is difficult to reproduce and debug?

Race condition is hard to reproduce and debug because it all depends on timing and that timing can be thrown off when you attempt to use a debugger or other methods. If the timing is different then you will not be able to emulate the race condition when debugging your program.

Question 3) How can it be fixed? Provide an example from your Project A_3.

Race condition can be fixed by avoiding it entirely by designing your software to make sure that the problem never arises in the first place. Or something could be implemented to make sure certain parts of code complete before moving onto other parts. An example from project A3 is to create a barrier that ensures that all the threads complete a certain section of code before anything other sections of code are processed. This then eliminates the possibility of race condition from happening due to the barrier for the threads.

Question 4) Summaries the Parallel Programming Patterns section in the "Introduction to Parallel Computing_3.pdf" (two pages) in your own words (one paragraph, no more than 150 words).

Developers use patterns that can be grouped into two main categories, 1) Strategies and 2) Concurrent Execution Mechanisms, when writing parallel programs. The two primary strategic considerations which should be considered are: 1) what algorithmic strategies to use, and 2) what implementation strategies to use when given the algorithmic strategies. For parallel programs, they often make use of several patterns of implementation strategies, and some of these patterns contribute to the overall structure of the program with others concerned with how the data that is being computed. To enable parallelism, or concurrent execution, two major categories, Process/Thread control patterns and Coordination patterns, are used. For the Coordination pattern, two popular C/C++ libraries, MPI and OpenMP, are often realized.

Question 5) In the section "Categorizing Patterns" in the "Introduction to Parallel Computing_3.pdf" compare the following:

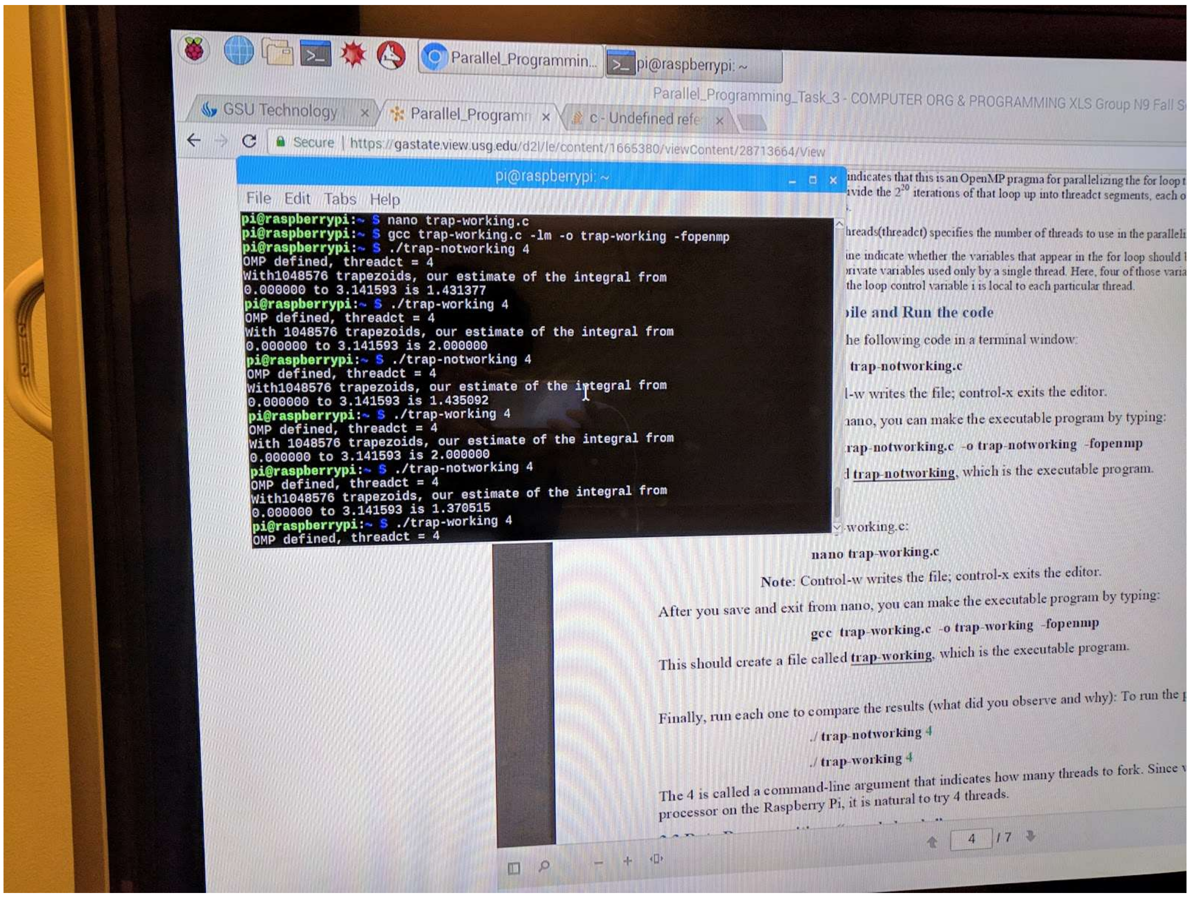
o Collective synchronization (barrier) with Collective communication (reduction)

Collective synchronization (barrier) and Collective communication (reduction) all belong to coordination, but they have different functions. Collective synchronization, or a barrier for a group of threads or processes in the source code, means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier. By contrast, Collective communication is a method of communication which involves participation of all processes in a communicator, but it must first implies the synchronization point.

o Master-worker with fork join

Master-worker and fork join all belong to program structure. But Fork-Join pattern is used to execute parallel processes (lightweight threads, not OS heavy threads). Master-worker pattern is used for parallel programming to divide a main process (master) into small chunks dispatched to several worker processes.

Lab Report



```
pi@raspberrypi:~ $ nano trap-working.c
pi@raspberrypi:~ $ gcc trap-working.c -lm -o trap-working -fopenmp
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.431377
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:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.435092
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:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.370515
pi@raspberrypi:~ $ ./trap-working 4
OMP defined, threadct = 4
```

indicates that this is an OpenMP pragma for parallelizing the for loop t
ivide the 2^{20} iterations of that loop up into threadct segments, each o

hreads(threadct) specifies the number of threads to use in the paralleli

ine indicate whether the variables that appear in the for loop should b
rivate variables used only by a single thread. Here, four of those varia
the loop control variable i is local to each particular thread.

ile and Run the code

he following code in a terminal window:

trap-notworking.c

l-w writes the file; control-x exits the editor.

nano, you can make the executable program by typing:

rap-notworking.c -o trap-notworking -fopenmp

d **trap-notworking**, which is the executable program.

working.c:

nano trap-working.c

Note: Control-w writes the file; control-x exits the editor.

After you save and exit from nano, you can make the executable program by typing:

gcc trap-working.c -o trap-working -fopenmp

This should create a file called **trap-working**, which is the executable program.

Finally, run each one to compare the results (what did you observe and why): To run the p

./trap-notworking 4

./trap-working 4

The 4 is called a command-line argument that indicates how many threads to fork. Since v
processor on the Raspberry Pi, it is natural to try 4 threads.

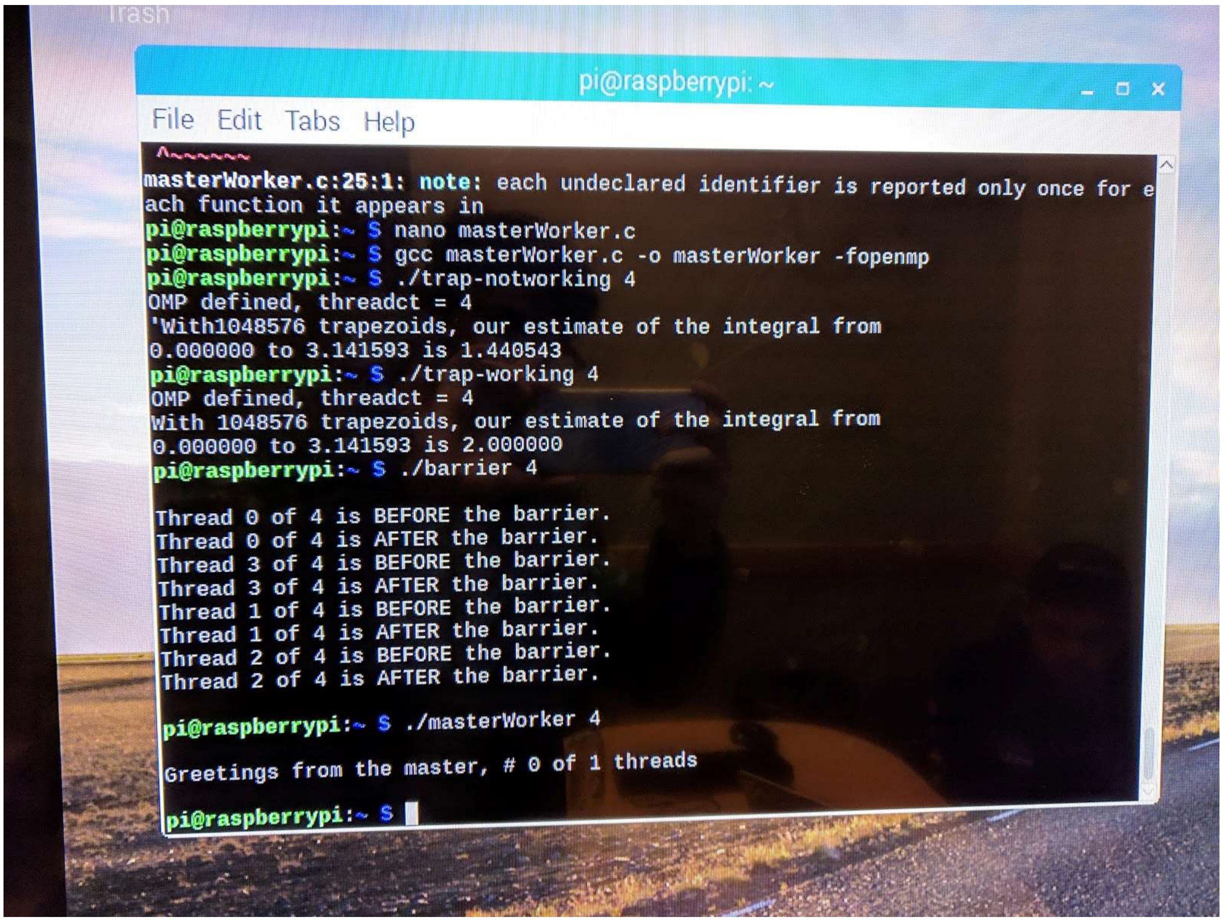
Figure 1: Integration/Coordination

In this figure it demonstrates what is asked In the first two parts of this task. Here within the code we are tasked with utilizing integration in efforts to find the volume. This was to be accomplished within a trapezoid, however we discovered that the two programs: notworking and working behaved differently. As shown here when trap-notworking was used it could not fully find the area and would be around 1.4 however trap-working was able to consistently reach 2.0 by using all of the threads available. The image demonstrates the how the clauses in the second line and how they are shared with the other threads and how all four are globally shared by all the threads.

```
pi@rasphberrypi:~$ ./trap-notworking 4

OMP defined, threadct = 4

With 10448576 trapezoids, our estimate of the integral from

0.000000 to 3.141593 is 2.000000

pi@rasphberrypi:~$ ./trap-working 4

OMP defined, threadct = 4

With 10448576 trapezoids, our estimate of the integral from

0.000000 to 3.141593 is 2.000000
```

This also shows the default behavior of consecutive iterations decomposing onto threads starting with 1048576 up to 2.

Fun Fact : What we noticed is that in order to make both of the programs executable they need -lm before the -o.



```
masterWorker.c:25:1: note: each undeclared identifier is reported only once for e
ach function it appears in
pi@raspberrypi:~ $ nano masterWorker.c
pi@raspberrypi:~ $ gcc masterWorker.c -o masterWorker -fopenmp
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
'With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.440543
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:~ $ ./barrier 4

Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 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.

pi@raspberrypi:~ $ ./masterWorker 4

Greetings from the master, # 0 of 1 threads

pi@raspberrypi:~ $ 
```

Figure 2: Synchronization with a Barrier/ The Master-Worker Implementation Strategy

The image shows our earlier results and how the barrier is used and ensures that all threads complete a parallel section of code before execution.Figure 2 also demonstrates how threads are shared within the program. We observed that in barrier 4 it used 4 threads also each one is used twice for each barrier and there is no repetition greater than 2.

The second portion of this task is focused on the master worker strategy which is one thread that will execute one block of code when it forks. This in a sense is a sort of hierarchy that will utilize the other threads commonly known as workers to execute the rest of the threads.

```
 //    #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)
```

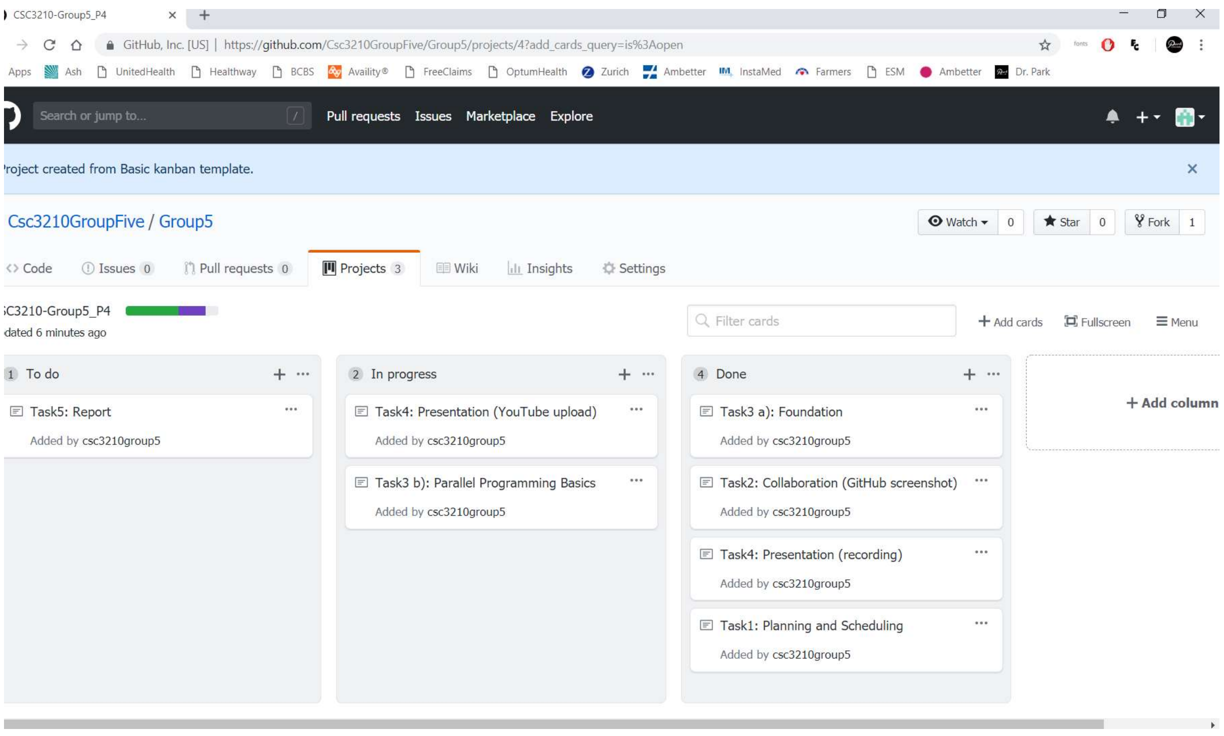
We end with the last requested task which is to demonstrate this implementation strategy with our raspberry pi. The above code is used and we demonstrate the result on the last line of the image Figure 2.

Appendix: Contact Links

Slack: https://csc3210group5.slack.com/archives/CCLF40G93/p1536007126000200

YouTube channel: https://www.youtube.com/channel/UCOP-InfUQx_cPpknqOH4cZA

Github: https://github.com/Csc3210GroupFive/Group5/projects/4


CSC3210-Group5_P4
https://github.com/Csc3210GroupFive/Group5/projects/4?add_cards_query=is%3Aopen
Pull requests Issues Marketplace Explore
Csc3210GroupFive / Group5
Code Issues 0 Pull requests 0 Projects 3 Wiki Insights Settings

roject created from Basic kanban template.

C3210-Group5_P4
dated 6 minutes ago

1 To do	2 In progress	4 Done
Task5: Report (Added by csc3210group5)	Task4: Presentation (YouTube upload) (Added by csc3210group5)	Task3 a): Foundation (Added by csc3210group5)
	Task3 b): Parallel Programming Basics (Added by csc3210group5)	Task2: Collaboration (GitHub screenshot) (Added by csc3210group5)
		Task4: Presentation (recording) (Added by csc3210group5)
		Task1: Planning and Scheduling (Added by csc3210group5)