Matthew Maloof

Project 4 Task 3

**Part A) Parallel Programming Skills**

**(2p) What is race condition?**

A race condition is the behaviour of software where the output is dependent on the sequence.

**(5p) Why is race condition difficult to reproduce and debug?**

Race conditions are difficult to reproduce and debug because the end result is nondeterministic where it depends on the relative timing between interfering threads. When running in debug mode, adding additional logs, or attaching a debugger can lead to these errors in the production systems disappearing.

**(8p) How can it be fixed? Provide an example from your Project\_A3 (see spmd2.c) -**

It is better to avoid race conditions by carefully programming your code, but to fix it, as shown in the example of spmd2.c in project 3, first you comment out the variable initializations at the top of your code then you can use full variable declarations (int id = omp\_get\_thread\_num();, int numThreads = omp\_get\_num\_threads(); vs int id, numThreads; at the top) which will make each thread have its own private copy of the variables leading to each thread being unique and only appearing once.

**(15p) 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). -**

There are two main categories of patterns that developers use for organized parallel code: Strategies and Concurrent Execution Mechanisms. Algorithm strategies and implementation strategies should be two considerations for developers. The concurrent execution pattern falls into two categories:

1. Process/thread control patterns dictates how the processing units of parallel execution are controlled at run time.

2. Coordination patterns sets up how the number of concurrently running tasks on the processing units coordinate for the parallel computation.

Message pathing is between concurrent processes on either single multiprocessor machines or clustered distributed computers. Mutual exclusion is between threads executing concurrently on a single shared memory space. Two examples of computation include MPI for message pathing and OpenMP for threaded, shared memory applications. A third involves hybrid computation using the two patterns above in a cluster of computers that executes multiple threads and uses both MPI and OpenMP in a single program.

**(12p) 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 blocks all the processes up to a specific synchronization point. This acts as a barrier with blocking the process until all other processes have reached that point. Collective communication involves all the processes reaching a specific point before the program executes. This acts as reduction because the process of the communicator collects the data from all the other processes and performs the operation to find the result.

**o Master-worker with fork join -**

The master-worker pattern involves the main process being split into smaller chunks which are then distributed to several worker processes where the fork join pattern is then used to execute parallel light-weight processes and threads.  
**(3p) Where can we find parallelism in programming?**

Parallelism can be found in any code that includes constraint by the sequence of operations needed to be performed for a correct output. This must address control, data, and system dependencies. If any of these do not occur, then parallelism cannot happen. Examples of where parallelism is used involves mobiles, databases, and servers.

**(6p) What is dependency and what are its types (provide one example for each)?**

A dependency is where an operation depends on an earlier operation to complete and produce an output before the later operation can be performed. The three types are true dependence, output dependence, and anti-dependence.

True dependence:

S1: a = 1;

S2: b = a; // S2 depends on S1

Output dependence:

S1: a = f(x);

S2: a = b; // S2 depends on S1

Anti-dependence:

S1: a = b; // S1 depends on S2

S2: b = 1;

**(3p) When a statement is dependent and when it is independent (Provide two examples)?**

If you have 2 lines of code, line 1 with statement 1 and line 2 with statement 2, they are considered independent of each other when statement 2 and statement 1 can switch lines and it does not affect the output of the program. If it does affect the output, then they are considered dependent on each other. 2 Examples below

S1: a=1;

S2: b=a; //Dependent (true (flow) dependence) where second is dependent on first.

--------

S1: a=1;

S2: b=1; //Independent, it doesn’t matter what order the 2 statements are in.

**(3p) When can two statements be executed in parallel?**

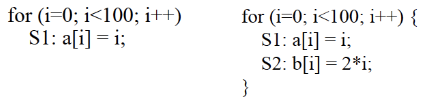
Statements S1 and S2 can be executed in parallel when there are no dependencies between S1 and S2. This includes true dependencies, anti-dependencies, and output dependencies.

**(3p) How can dependency be removed?**

A dependency can be removed by modifying the program either by rearranging statements or eliminating statements.

**(8p) How do we compute dependency for the following two loops and what type/s of**

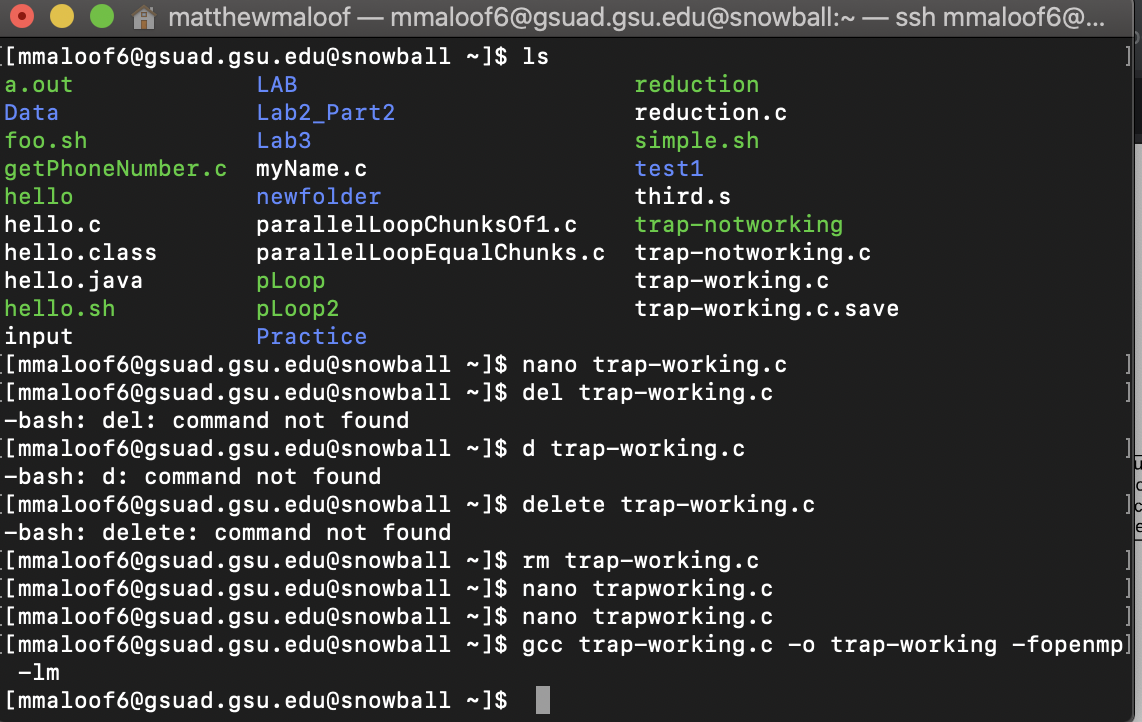
**dependency?**

****

In the first loop, in “S1: a[i] = i;”, a[i] is dependent on variable i, so it needs to be placed after the statement where i is initialized. This dependency type is True dependency because the order matters.

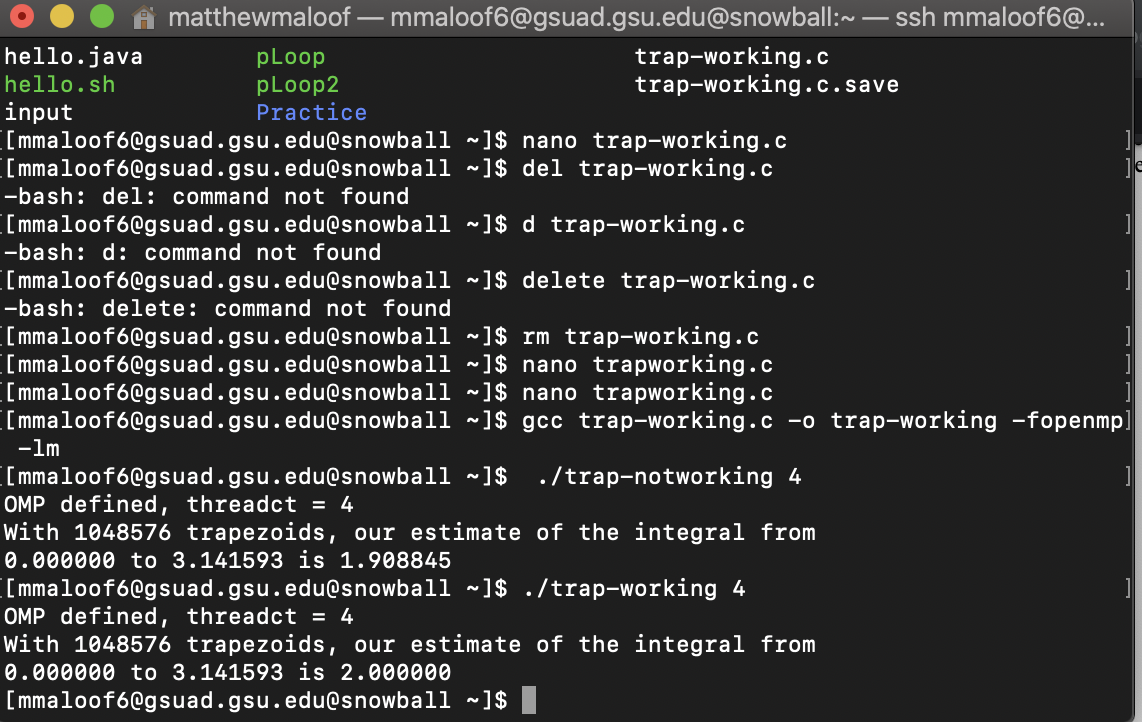
In the second loop, in “S1: a[i] = i;”, a[i] is again dependent on variable i, so it needs to be placed after the statement where i is initialized. This type is also True dependency with the order mattering. Also, in “S2: b[i] = 2\*i;”, b[i] is dependent on i with True dependency (order matters).

**Part B) Parallel Programming Basics**

****

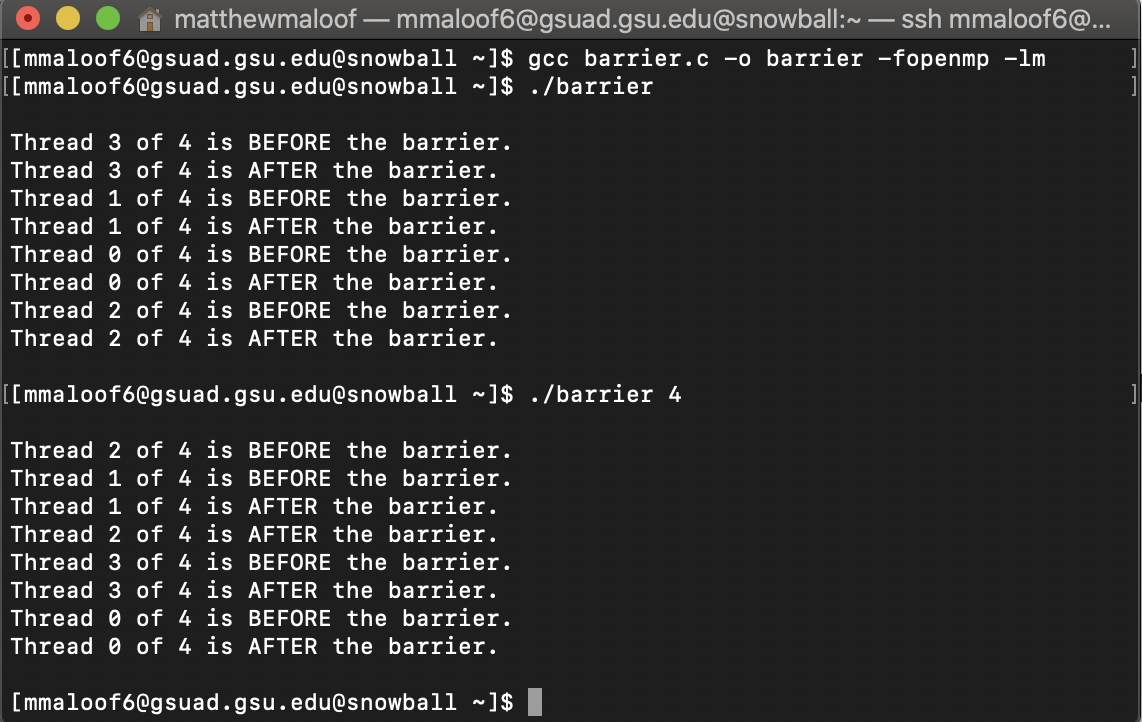
**Figure 1**

First, I copied the blocks of codes on the slides and put them into trap-notworking.c and trap-working.c respectively using nano command. Then I turned them into executables using gcc command.

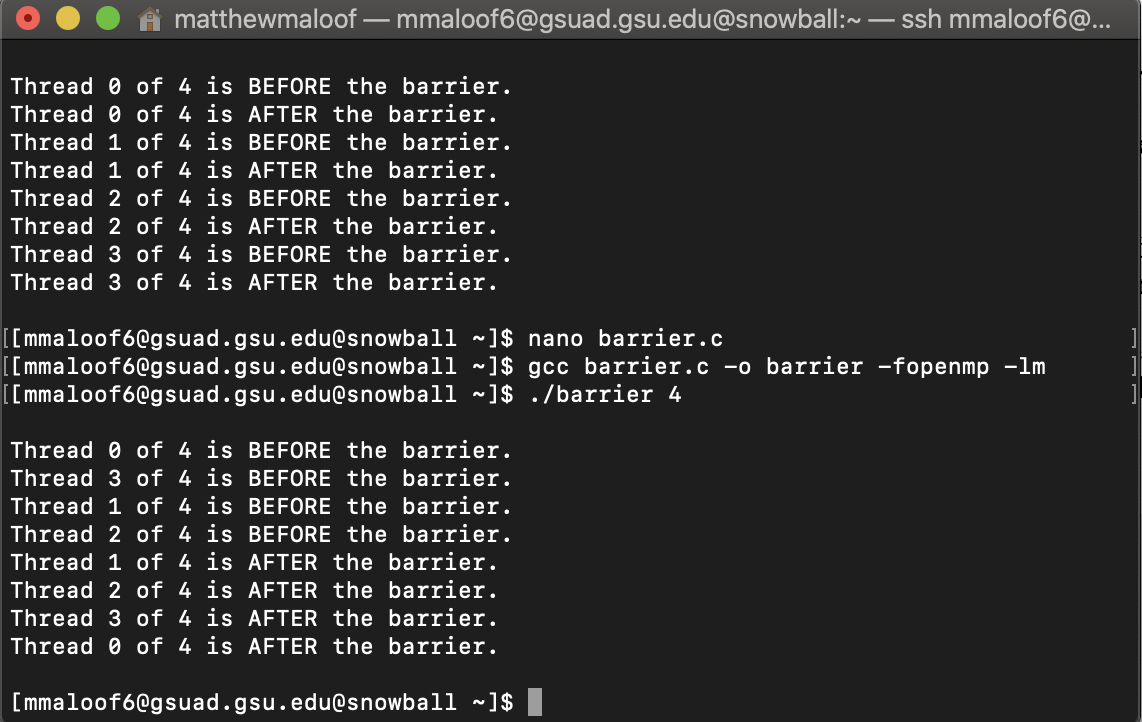


**Figure 2**

Then, I type ./trap-notworking 4 and ./trap-working 4 to run the programs with 4 threads to fork. The outputs are above. As we can see, there are 2 different outputs with one being 1.908845 and the working program outputting 2.000000. This different output is shown because of the change in code in line 37 from the not working to the working program. This change is that instead of having #pragma omp parallel for private(i) shared (a,n,h,integral) on one line, we split it between for and private(i) into two lines, and added a ‘\’ after ‘for’. Having no explicit addition of an additional static or dynamic clause to the pragma for assigning equal amount of work using consecutive iterations of the loop used the default behaviour to decompose the problem onto threads.

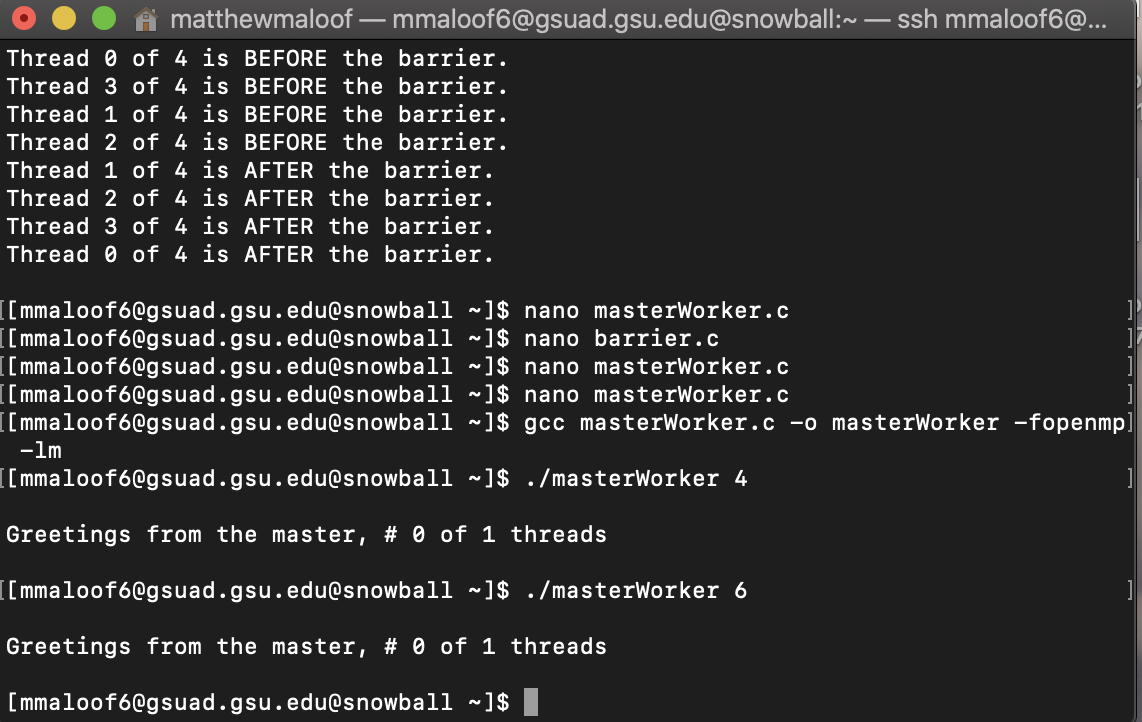
  
**Figure 3**

A new file was created using nano barrier.c and the code in the slide was copied and pasted. Then I used gcc to turn it into an executable and running the program with 4 threads to fork shows the output above. This output is with the program having commented out pragma on line 31.



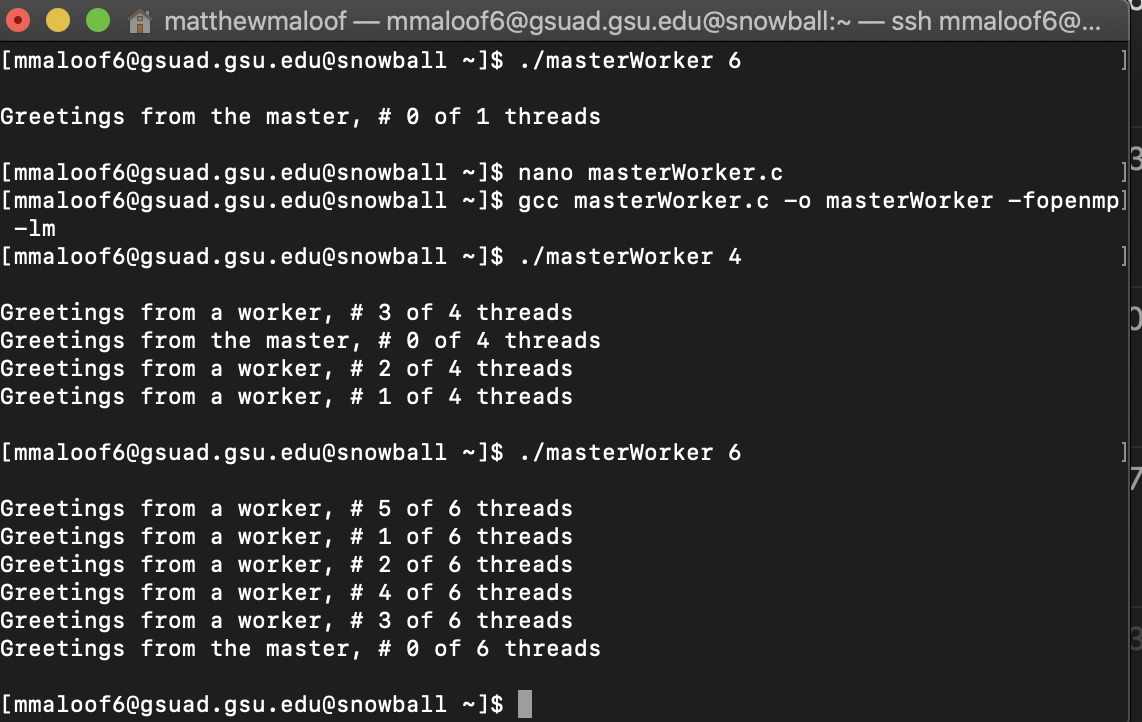
**Figure 4**

After uncommenting the line 31 containing pragma omp barrier, the program first prints out the before barrier lines of code first then starts printing after the barrier, which should happen because the barrier makes all threads before it reach the same point then it goes past the barrier. The barrier is between the before and after print statements, so this makes sense.



**Figure 5**

I created a new file using nano called masterWorker.c. I then copied and pasted the code on the slide and made the file executable using gcc. The output for masterWorker program with the pragma line 24 commented out is shown above.



**Figure 6**

After uncommenting out line 24 that contained the pragma directive, I recompiled the program and the output is shown above. The difference is that with the pragma commented out it just showed the master thread count which always starts at 0. With the pragma uncommented, the worker threads are also being shown which is always from 1 to thread total - 1 along with the master thread which is always shown at thread 0 of total. The reason this is the output is because without the pragma directive, the code just runs once and that’s the end, and since it runs once the master thread will always be at the 0 count so it is the only one that gets outputted. With the pragma directive, it will keep printing out the other threads labeled as worker threads until it reaches the thread total - 1 thread.