Task 3

1. Foundation

**What is race condition?**

* A race condition is when a functionality of electronics or software is dependent on the timing of an uncontrollable event and may cause issues if the timing is different than what the Programmer anticipates and could potentially turn into a difficult to remedy bug.

**Why is race condition is difficult to reproduce and debug?**

* They are difficult to reproduce and debug because they are dependent on the timing between many items of running code. Additionally, since additional logging is added in debugging mode sometimes the problem can disappear when the problem is trying to be replicated, which makes it very difficult to debug.

**How can it be fixed? Provide an example from your Project\_A3**

* A race condition can be fixed by declaring the variables inside the pragma. That way each thread can have their own private copy of the variables. In Project 3 (spmd.2.c), we declared the variable id inside the pragma to create separate id for each thread.

**Summarize the Parallel Programming Patterns section in the “Introduction to Parallel Computing\_3.pdf”**

* We can distinguish two main categories of patterns for parallel programming: Strategies, or the general programmer-centric implementations of code, that take different forms in different systems, but ultimately achieve the same results across the board, and concurrent execution mechanisms, or patterns unlike strategic patterns in the sense that they are not general in nature and are closely aligned with the systems that they operate within. Each strategic pattern can be broken down into algorithms and implementation choices while concurrent execution mechanisms fall squarely into process/thread control patterns or coordination patterns. With new technologies being created regularly, newer implementations are being synthesized for increased capability and efficiency of systems, which has ultimately led to an increasingly married hybrid computation, consisting of the two pattern types operating on a cluster of computers.

**In the section “Categorizing Patterns” in the “Introduction to Parallel Computing\_3.pdf” compare the following:**

* Collective synchronization(barrier) vs Collective communication(reduction)

Collective synchronization using barriers to control the statements being executed by single thread or multiple threads. It means that all threads cannot proceed until other threads reach the same point. Collective communication using reduction to combine multiple threads through combining trees or other structures. It also lets you specify private variable specific to this thread that are subject to reduction operation.

* Master-worker vs fork join

Master-worker model working on one thread, when it runs one block of code which will fork, then we call this block of code is master, the rest of block are workers. Which means even inside the same thread, we still execute different operation, then combine the result as the result of this thread. Fork-join model is similar, execution branches off in parallel at designated points in the program, to "join" (merge) at a subsequent point and resume sequential execution. Fork-join can be a design pattern, and a common pattern using fork-join to deal with multiple data is master-work.

**Where can we find parallelism in programming?**

* In the Program(task) view, the Data view, and the Resource view. In Program(task) view we can find it within a Statement level or Block, Loop, Routine or Process level.

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

* Dependency is when an operation relies upon another operation that much be completed prior to the execution of said operation.
* The types of dependency are:
  + - True Dependencies
      * Second is dependent on the first
      * Example:
        + S1: a= 5
        + S2: b = a
    - Output Dependencies
      * Second is dependent on the first
      * Example:
        + S1: a= g(x)
        + S2: a= b
    - Anti-Dependent
      * First is dependent on the second
      * Example:
        + S1: a = b
        + S2: b = 1

**When a statement is dependent and when it is independent? (examples)**

* A statement is dependent when it relies upon another separate statement in order to work.
  + S1: a= 5
  + S2: b = a
* A statement is independent when it can run by itself without having to rely upon any other separate statements.
  + S1: a = 5
  + S2: b = 3

**When can two statements can be executed in parallel?**

* Statements can only be executed in parallel if neither is dependent on the other in order to execute.

**How can dependency be removed?**

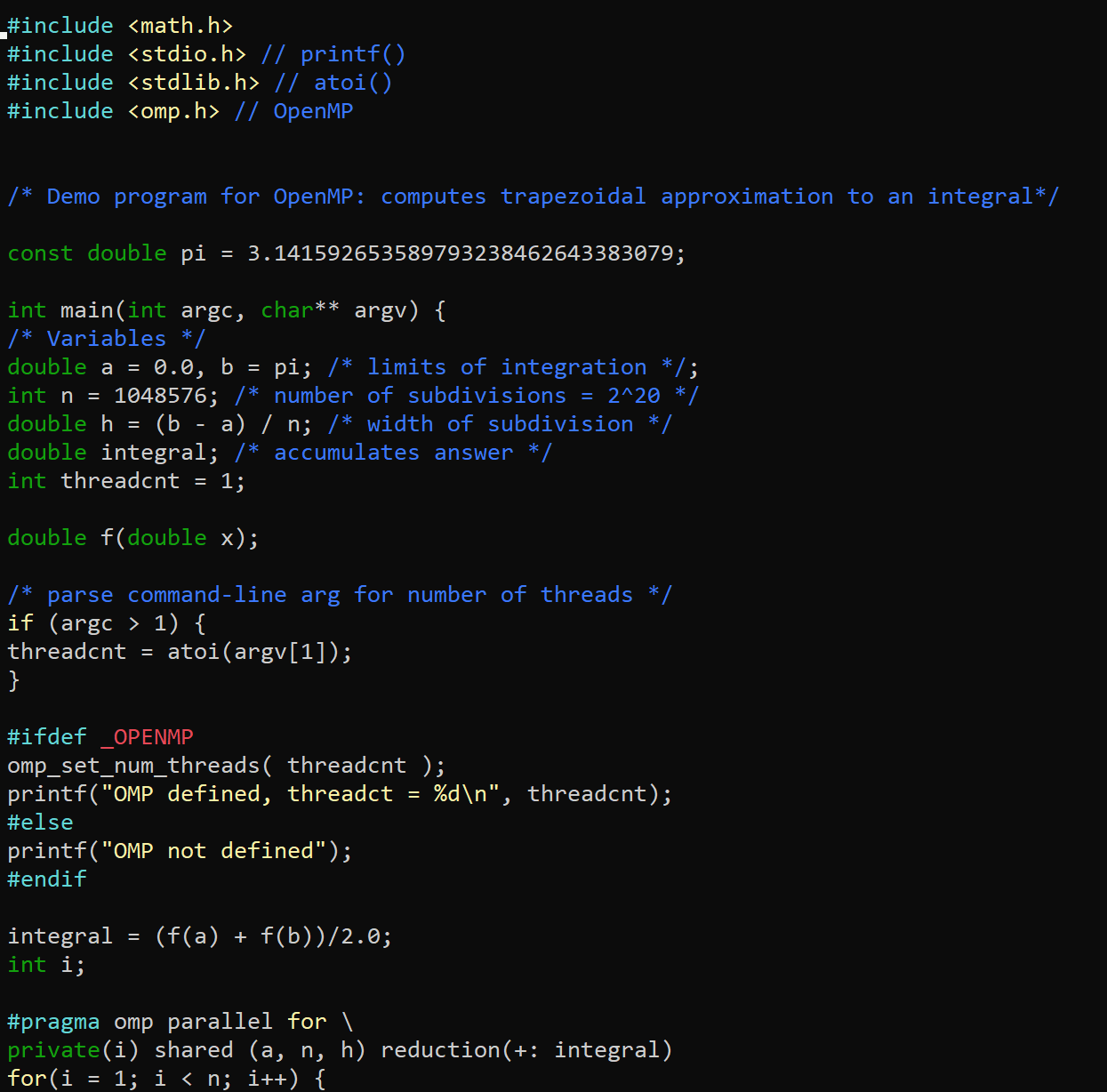
* We can eliminate or rearrange the statements of the program to remove the dependency.

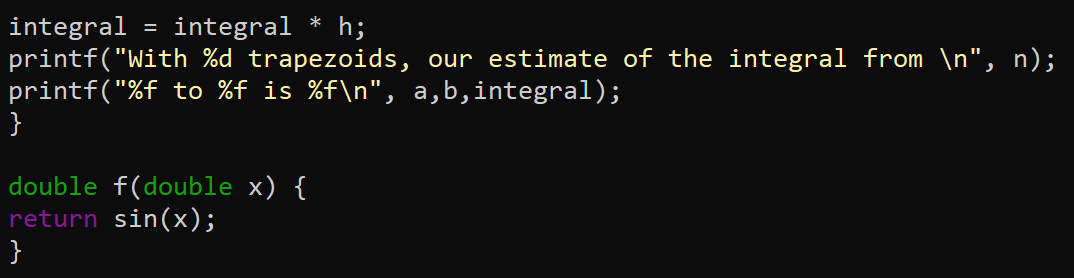
**How do we compute dependency for the following 2 loops and what are the type/s of dependency?**

* We compute dependency by looking at the IN(the memory variables that can be used in S) and OUT(memory variables that can be changed by S) sets of a given statement S
  + for(i=0, i<100,i++){
    - S1: a[i]=i;
  + This is an example of True Dependency because it is reliant upon the value of *i* to dictate the output.
  + for(i=0, i<100,i++){
    - S1: a[i]=i;
    - S2: b[i]= 2\*i;
  + This is an example of Output Dependency because the value of b[i] is being changed by (2\*i) and is the result of a function.

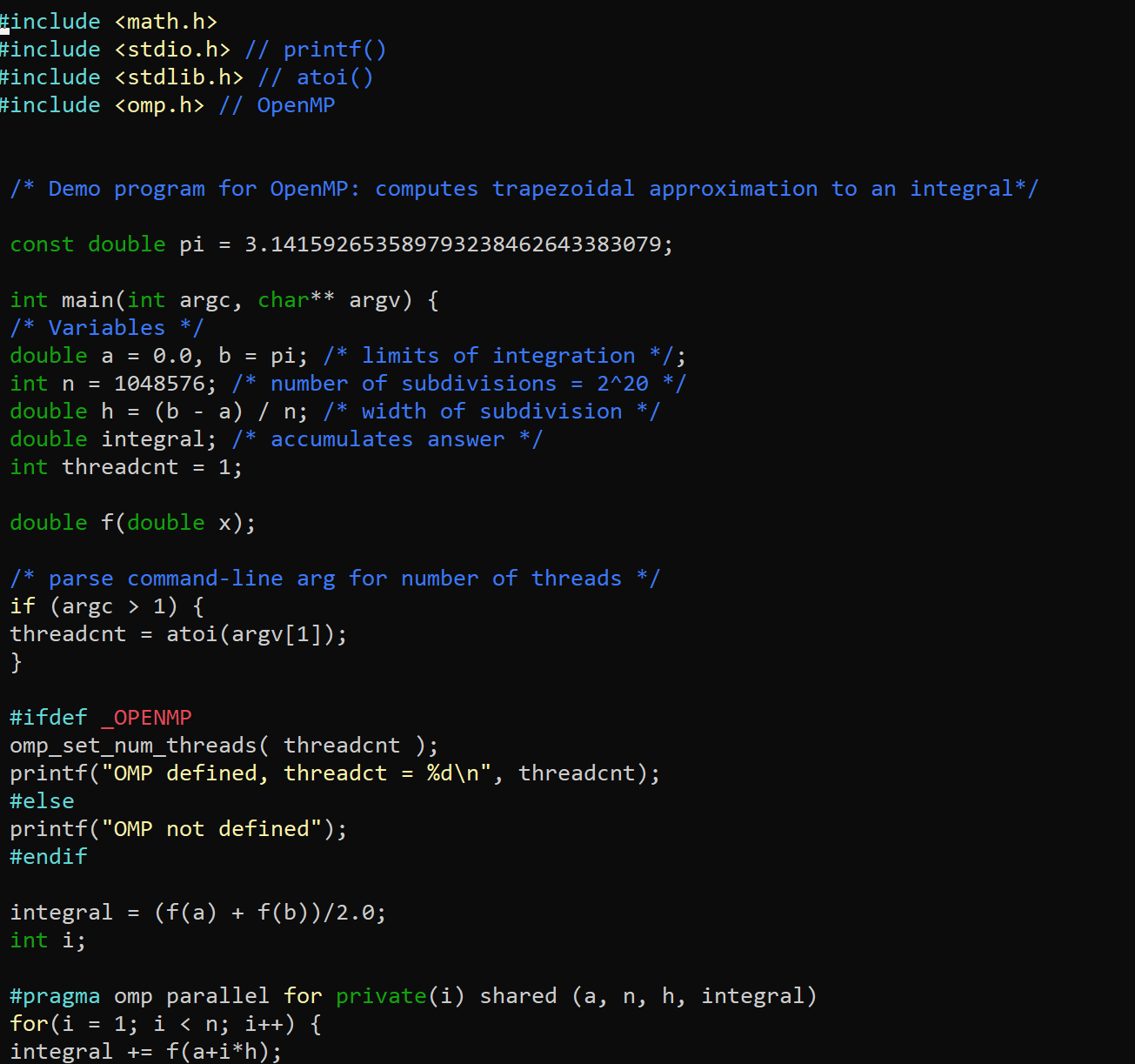
1. Parallel Programming

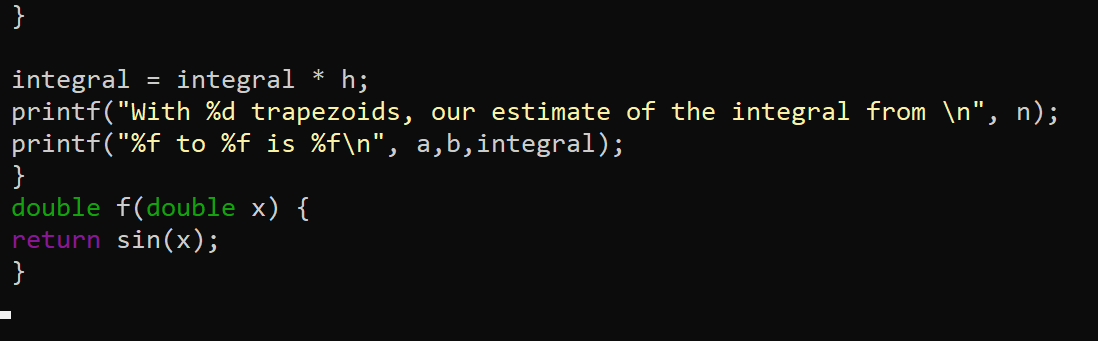
I copied and pasted the given code for trap-working.c (Figure 1) and trap-notworking.c (Figure 2) programs, made them executable files and ran them (Figure 3). Since the code imported math library, I had to add “-lm” to the gcc command while compiling it. The answer for the computation for trapezoidal approximation of using 220 equal subdivisions should equal to 2.0. However, the output of the trap-notworking file is 1.423339 not 2.0 like it is in trap-working file. The difference between these programs is the fact that in trap-working program reduction operator for the integral variable was used.





*Figure 1 – trap-working program*





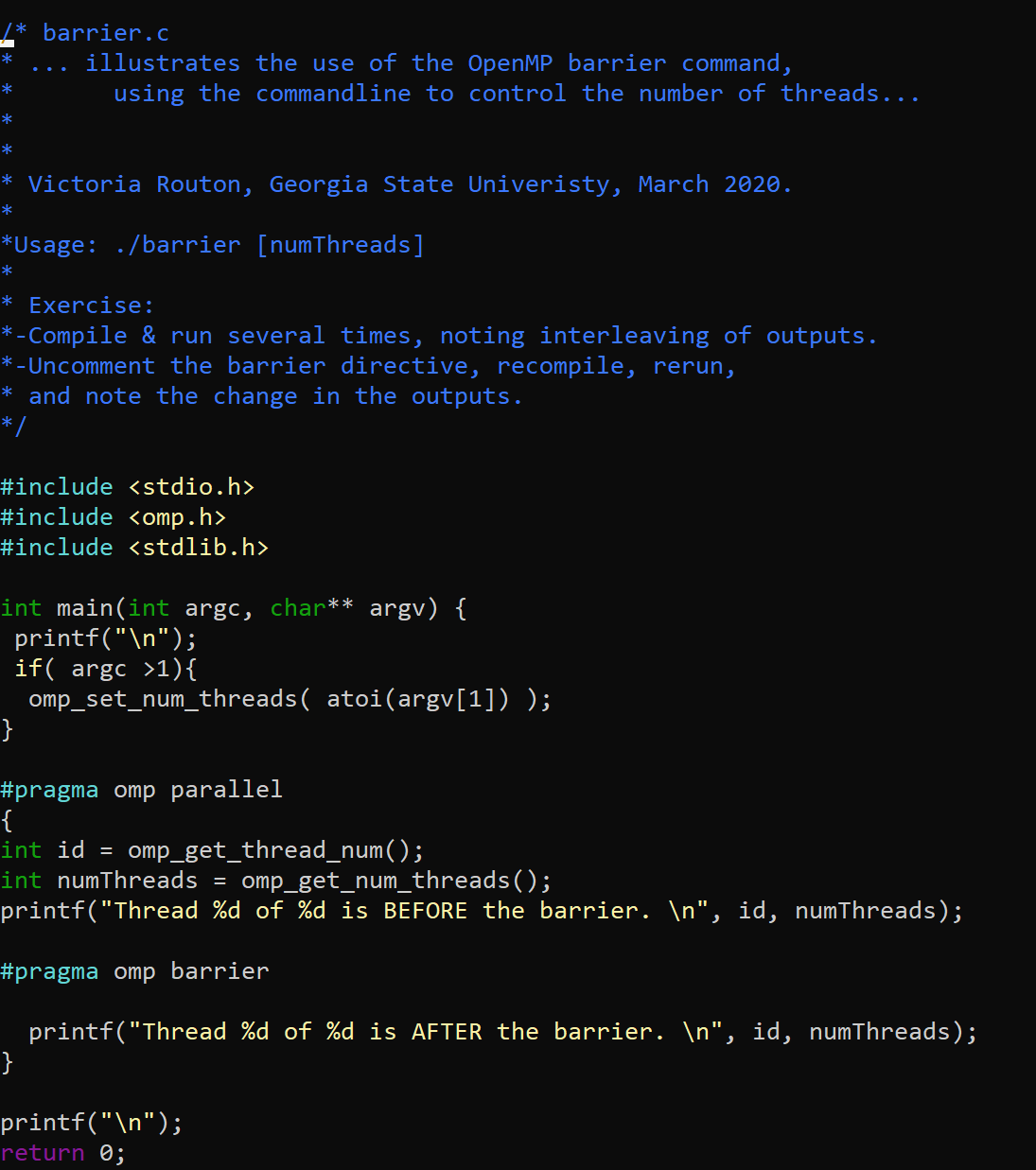
*Figure 2 – trap-not-working program*

A screenshot of a cell phone

Description automatically generated

*Figure 3 - running both trap-notworking and trap-working programs*

I copied and pasted a barrier code (Figure 4), compiled it and ran it. I notice that it prints a “BEFORE” and “AFTER” for each thread (Figure 5)



*Figure 4 – barrier code*

*A screenshot of a computer

Description automatically generated*

*Figure 5 – running the barrier program*

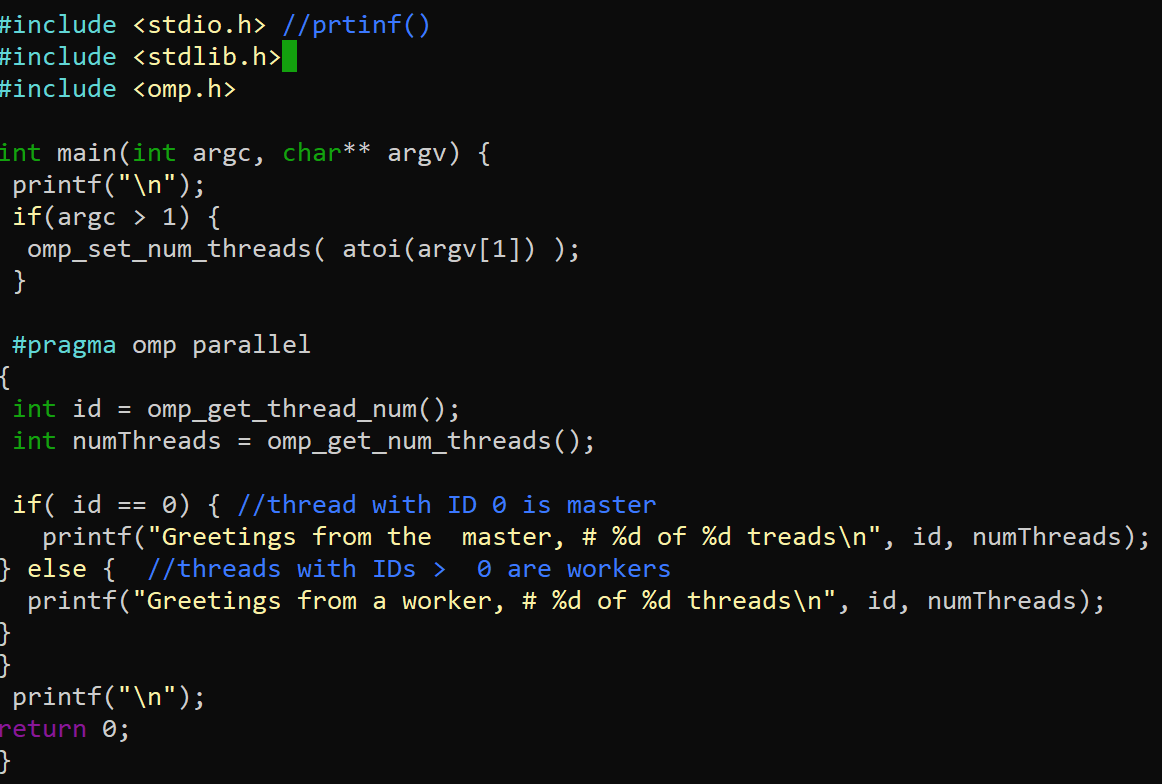
After uncommenting the “#pragma omp barrier”, I compiled and ran the code (Figure 6). I noticed that in this case, it prints all “before” lines of code before it moves to “after”. This outcome is to be expected due to the changes I made by uncommenting the pragma part, the threads meet the same point and then they all move to “after” lines of code.

A screenshot of a computer

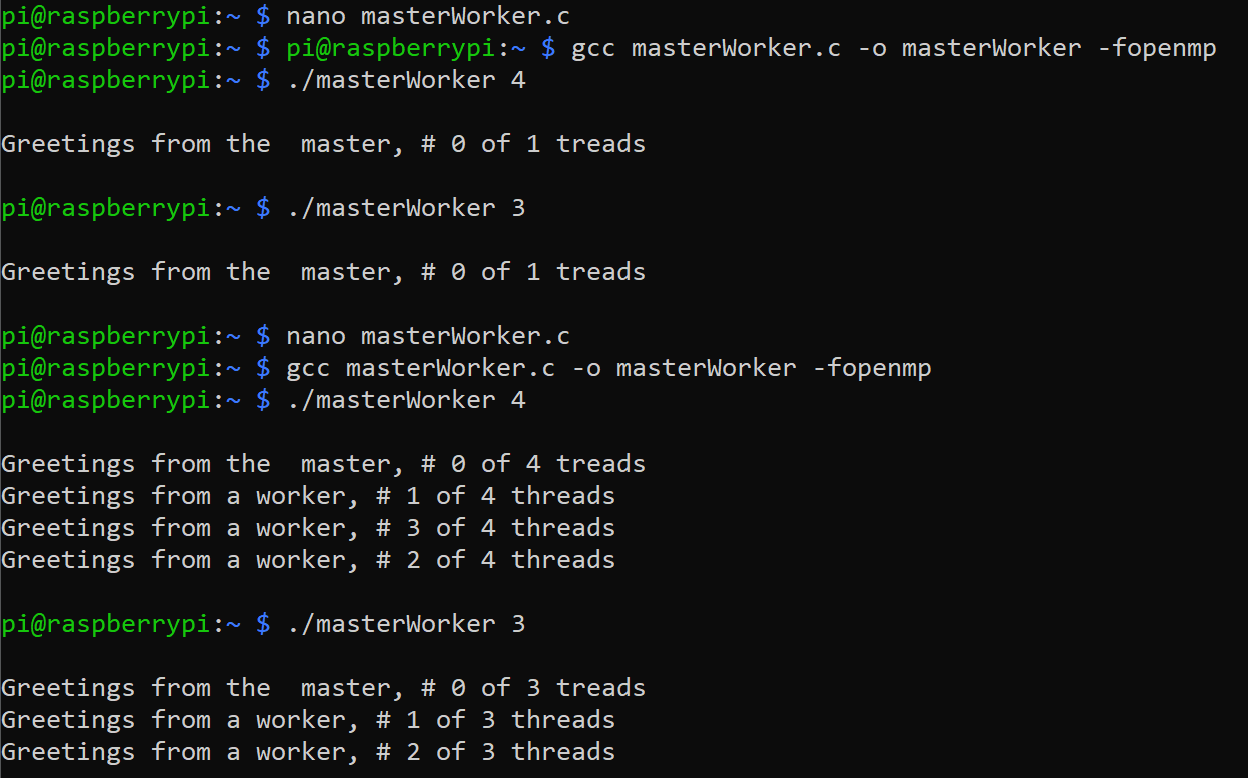
Description automatically generated

*Figure 6 – barrier code after uncommenting the pragma part*

Then, I made masterWorker code (Figure 7), compiled it and ran it. The outcome that I got was displayed below. Then, I uncommented the #pragma part, compiled and ran the program again to see what the differences I will get (Figure 8).



*Figure 7 – masterWorker code*



*Figure 8 - running masterWorker*

Before uncommenting “#pragma omp parallel”, I noticed that the program just went through first thread (masters) then stopped, which caused not to execute the workers. After uncommenting “#pragma omp parallel”, both master and workers block of code are being executed and the program produces the correct number of threads that I set.