Parallel Programming Skills

1. Foundation

Race condition:

*What is a race condition?*

A race condition occurs within software or systems when there is a dependency on the sequence of other events beyond the programmer’s control that affects the output.

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

Race conditions are hard to reproduce because often, the ending results or output will vary despite using the same input and conditions, since it depends on the timing of the threads.

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

You can fix this issue by having good software architecture to prevent the issue from occurring in the first place. An example where the issue is fixed beforehand comes from Assignment 3’s Parallel Programming task. The parallel for pragma without the reduction clause resulted in an incorrect outcome, but the solution to this problem simply required declaring the variables as private so that each thread had its own clone. This fixed the issue because the sum variable was dependent on each of the threads’ processes during computation.

Summarize 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 programs have patterns which are used often by programmers due to their success in previous programs. These patterns are documented for new programmers to use. There are two major categories for the patterns that developers use: strategies and concurrent execution mechanism. Strategies simply refer to the programmer’s choice on which “algorithmic strategies” to use as well has how to implement those strategies. Concurrent execution mechanisms have two major categories as well: process/thread control patterns and coordination patterns. Coordination patterns can further be separated into message passing and mutual exclusion. Process/thread control patterns involve the processing units and how they are executed at runtime whereas coordination patterns involve determining how processing units coordinate to complete tasks.

Compare collective synchronization (barrier) with Collective communication (reduction) and Master-worker with fork join.

Barriers are used to enforce collective synchronization by adding wait cycles into a program where a process cannot continue until the other processes have reached the same point, where they will be synchronized. Reductions are used to enforce collective communication by combining multiple vectors of processes or processes into one. Both patterns are concurrent execution mechanisms.

A Master-Worker pattern uses a master process/thread to hold a group of worker processes with a certain number of tasks. The “workers” complete each task simultaneously while removing them from the tasks held by the master process. The Fork-Join pattern describes the process of forking parent tasks to create new tasks and completing them concurrently with other forked tasks, then waiting until they finish executing before joining them further continue the parent task. Both patterns are strategies.

Dependency:

*Where can we find parallelism in programming?*

You can find them in loops!

*What is dependency and what are its types?*

A dependency occurs when the execution of one operation is reliant on the outcome of a previous operation and its execution. There are two types of dependencies—independent and dependent.

Independent:

Statement 1 – a = 3;

Statement 2 – b = 3;

Dependent:

Statement 1 – a = 3;

Statement 2 – b = a;

*When a statement is dependent and when it is independent?*

A dependency is independent if changing the order of execution does not affect the outcome. Otherwise, if changing the order causes an error or results in an undesirable outcome, then the dependency is dependent.

Independent:

Statement 1 – a = 33;

Statement 2 – b = 33;

Dependent:

Statement 1 – a = b;

Statement 2 – b = 33;

*When can two statements be executed in parallel?*

Two statements can be executed in parallel when the order of the operations do not affect the outcome and the statements themselves, do not have any dependencies.

*How can dependency be removed?*

Dependencies can be removed by changing the order of the statements or removing some of them completely.

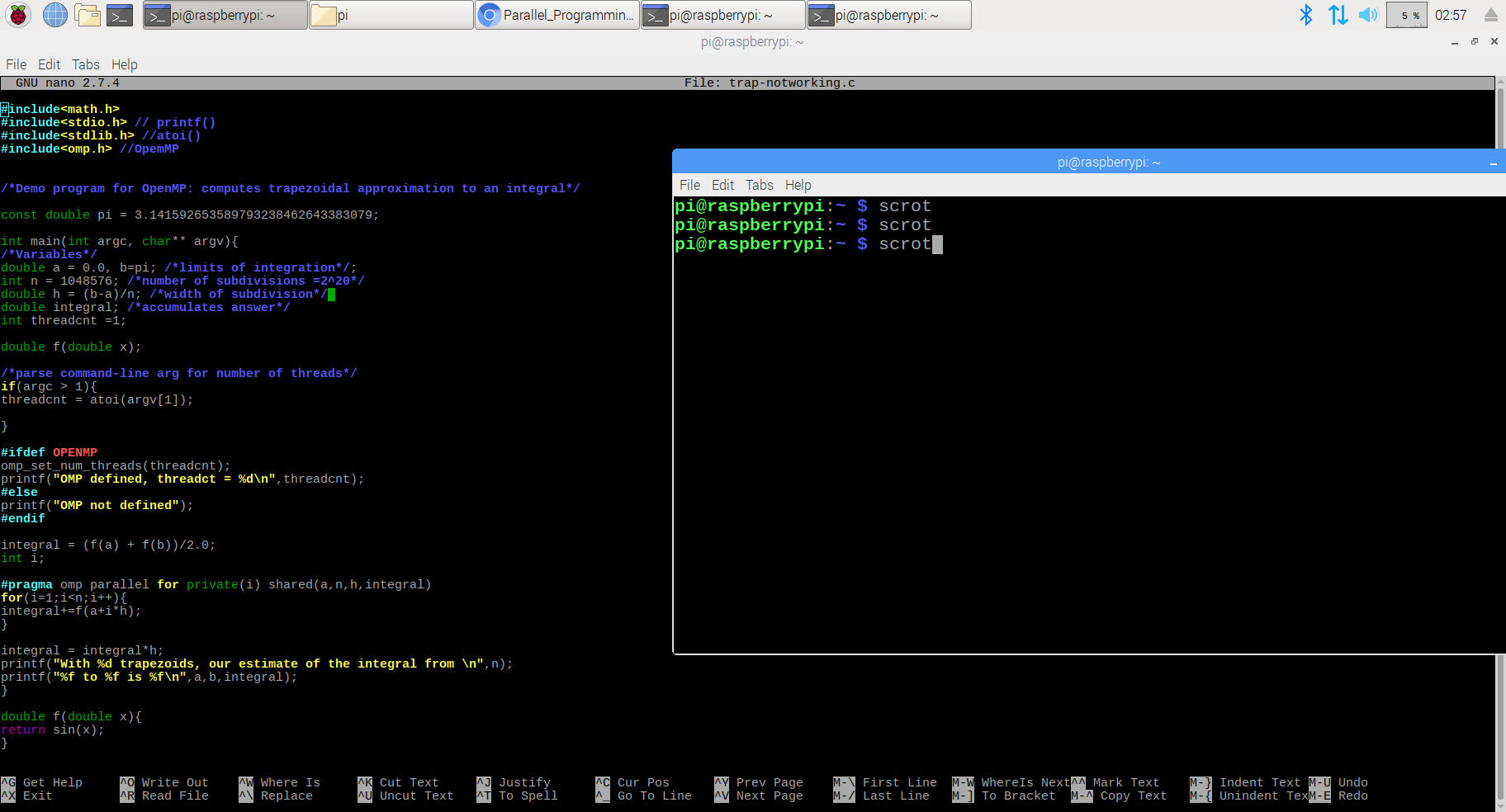
*How do we compute dependency for the following two loops and what type/s of dependency?*

The two loops are dependent on the value of i in order to find the correct index of a and set the correct value to it.

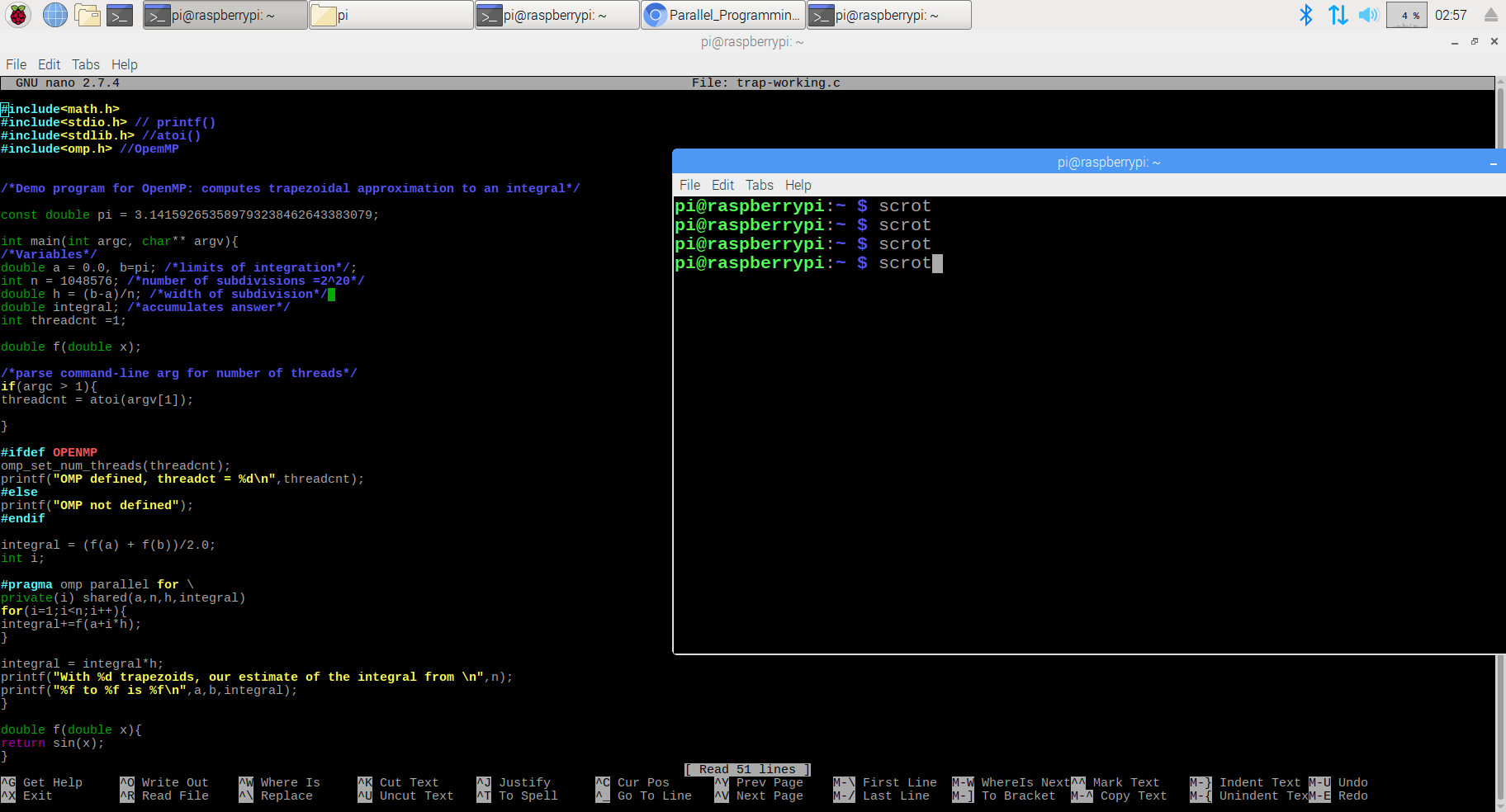
1. Programming

**Part One: Integration Using the Trapezoidal Rule**

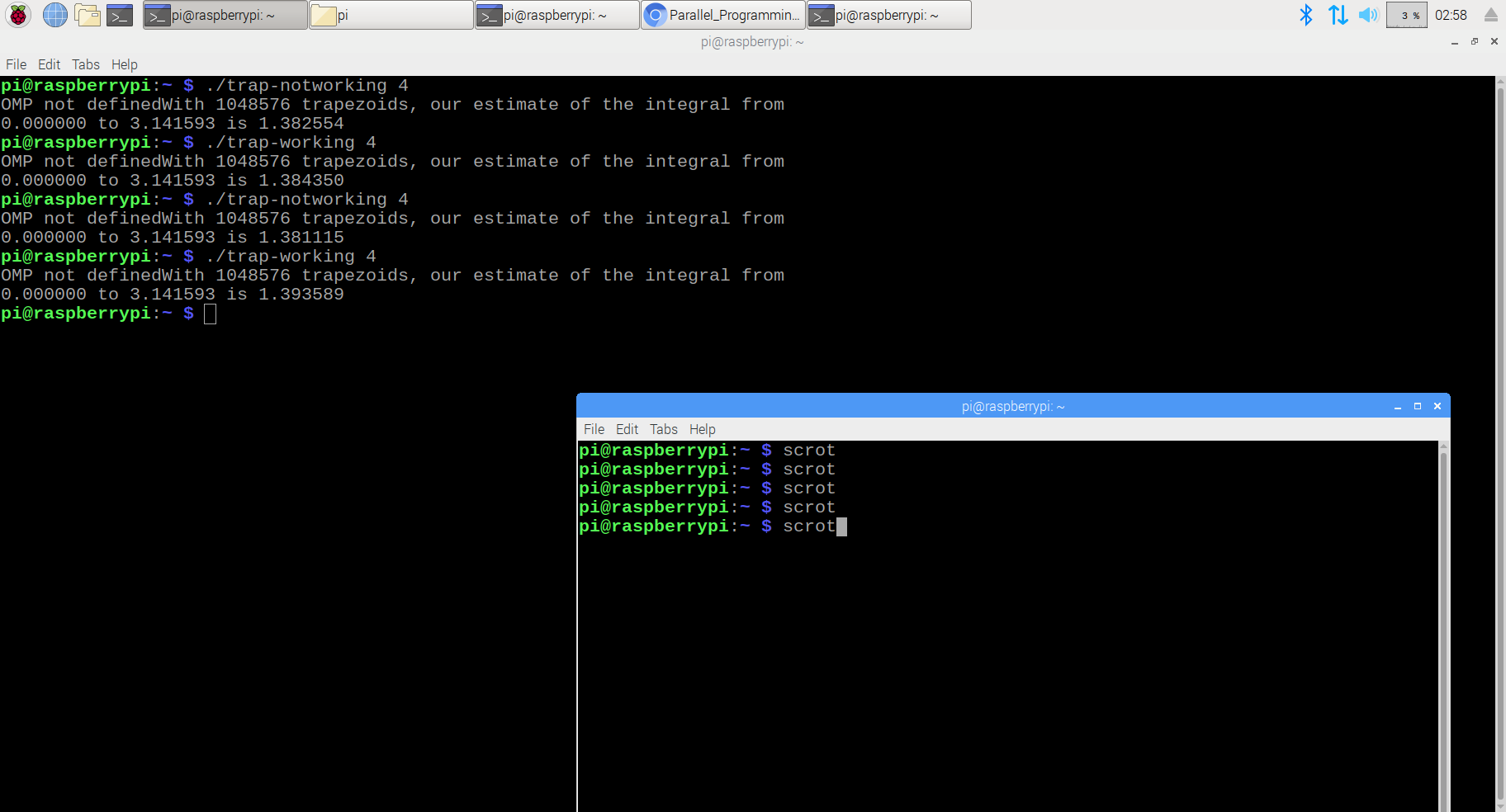
For the first part of this section, I created a file called trap-notworking.c using nano and copied he code provided on the pdf file. I then created an executable as instructed.



After that, I created a new file called trap-working.c and corrected the error by adding a backslash after “#pragma omp parallel for” and moving the line with “private” to the next line. I then created another executable.

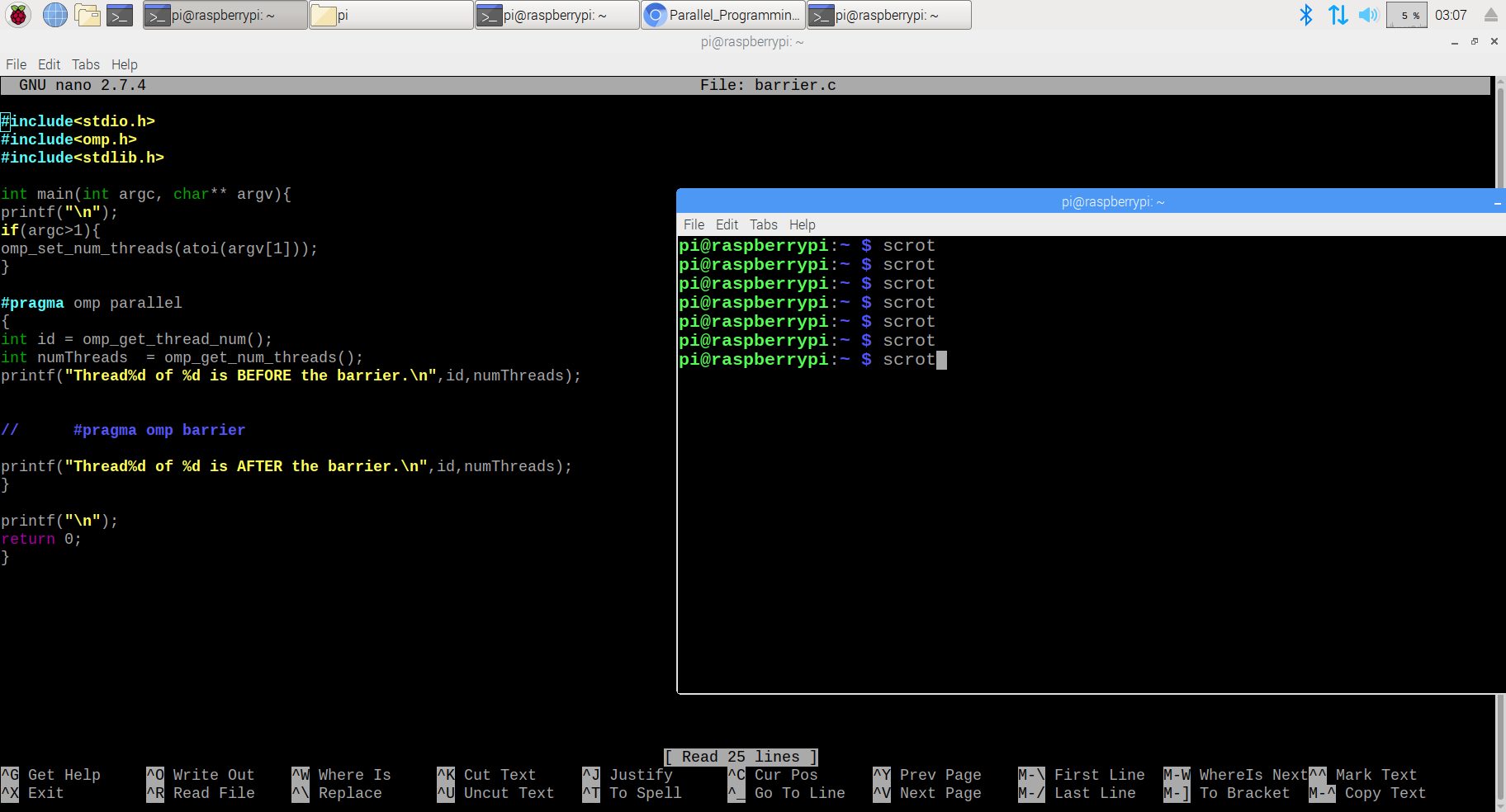


I compared the results between two of the programs multiple times to check if the intended result was reached.

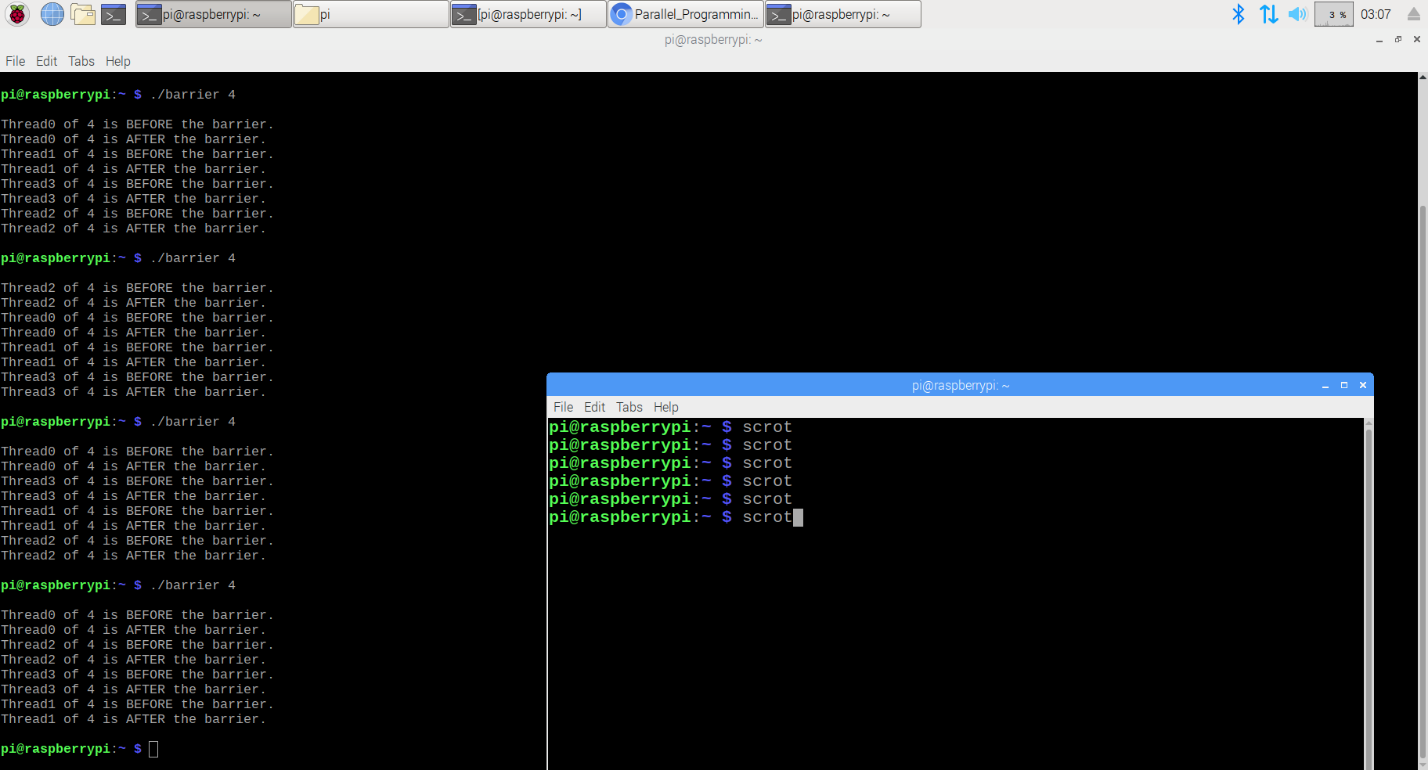


**Part Two: Coordination: Synchronization with a Barrier**

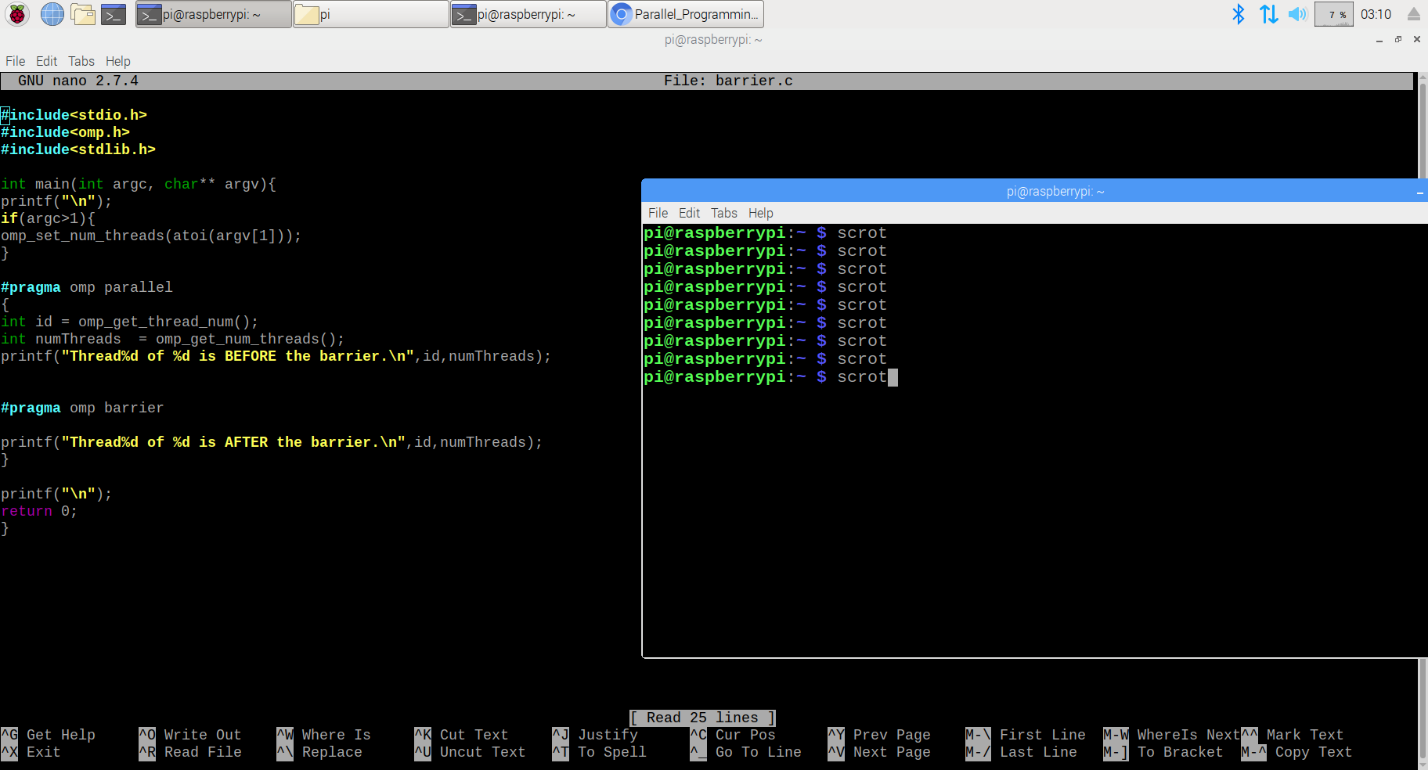
For this section of the task, I read the short excerpt on barriers and then proceeded to complete the programming portion. I created a barrier.c file and copied the code from the pdf.



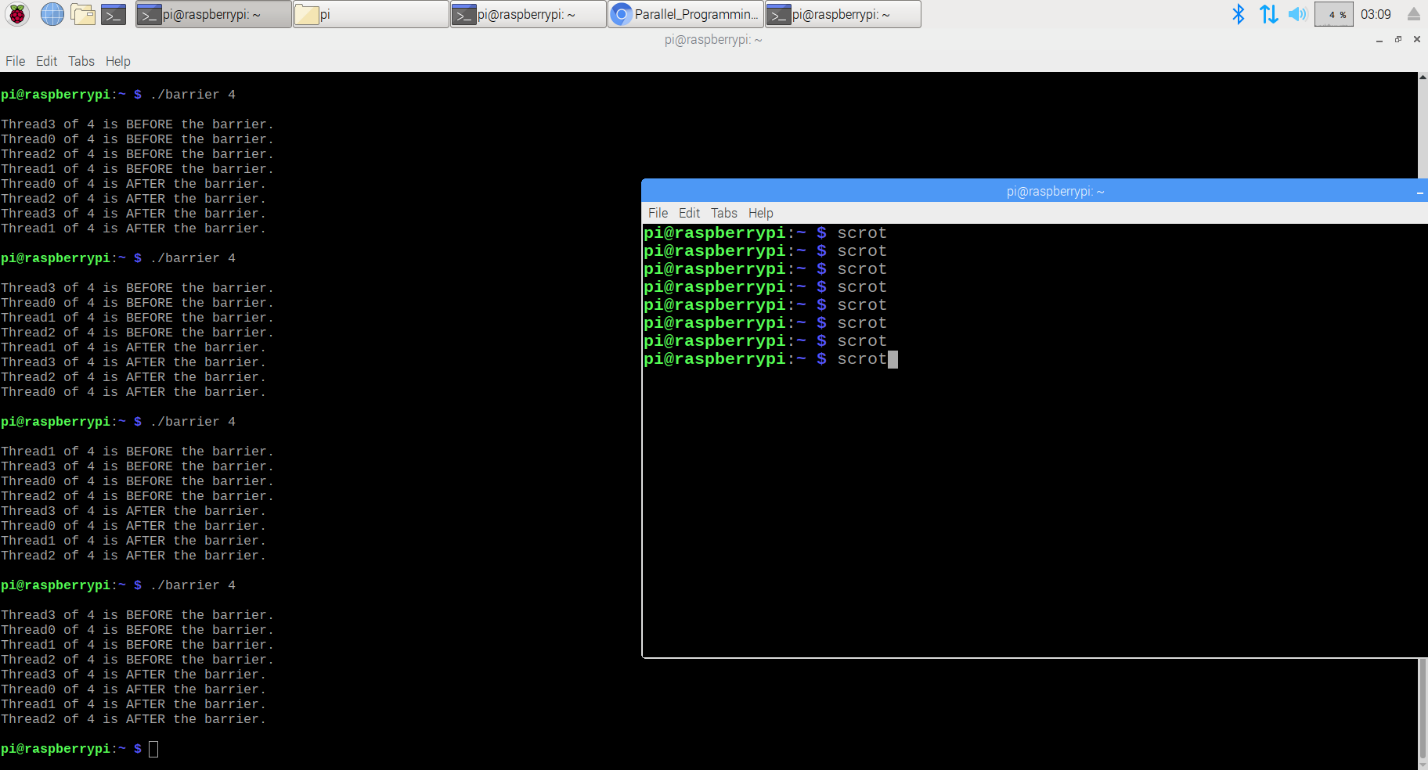
I created an executable for this program and ran it with the “#pragma omp barrier” line still commented out.



As you can see, with the barrier not effect, the program simply runs the operations in random, but sequential order, one after another without stopping. Continuing the task, I uncommented the “#pragma omp barrier” line and ran the program once again.

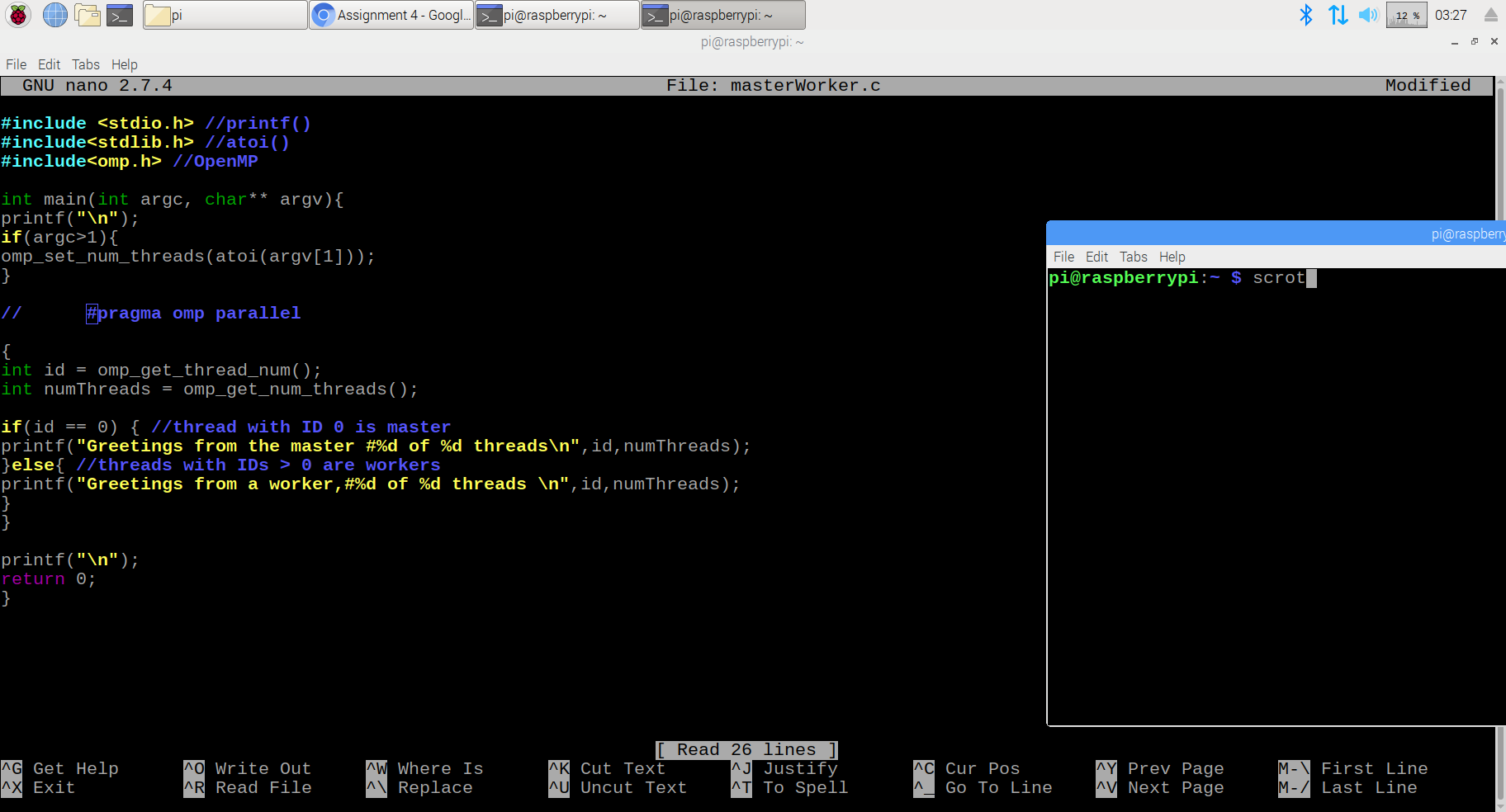


With the barrier in effect this time, you can see that the threads stop before the barrier and continue executing together after the barrier, in contrast to what we observed earlier without the barrier.

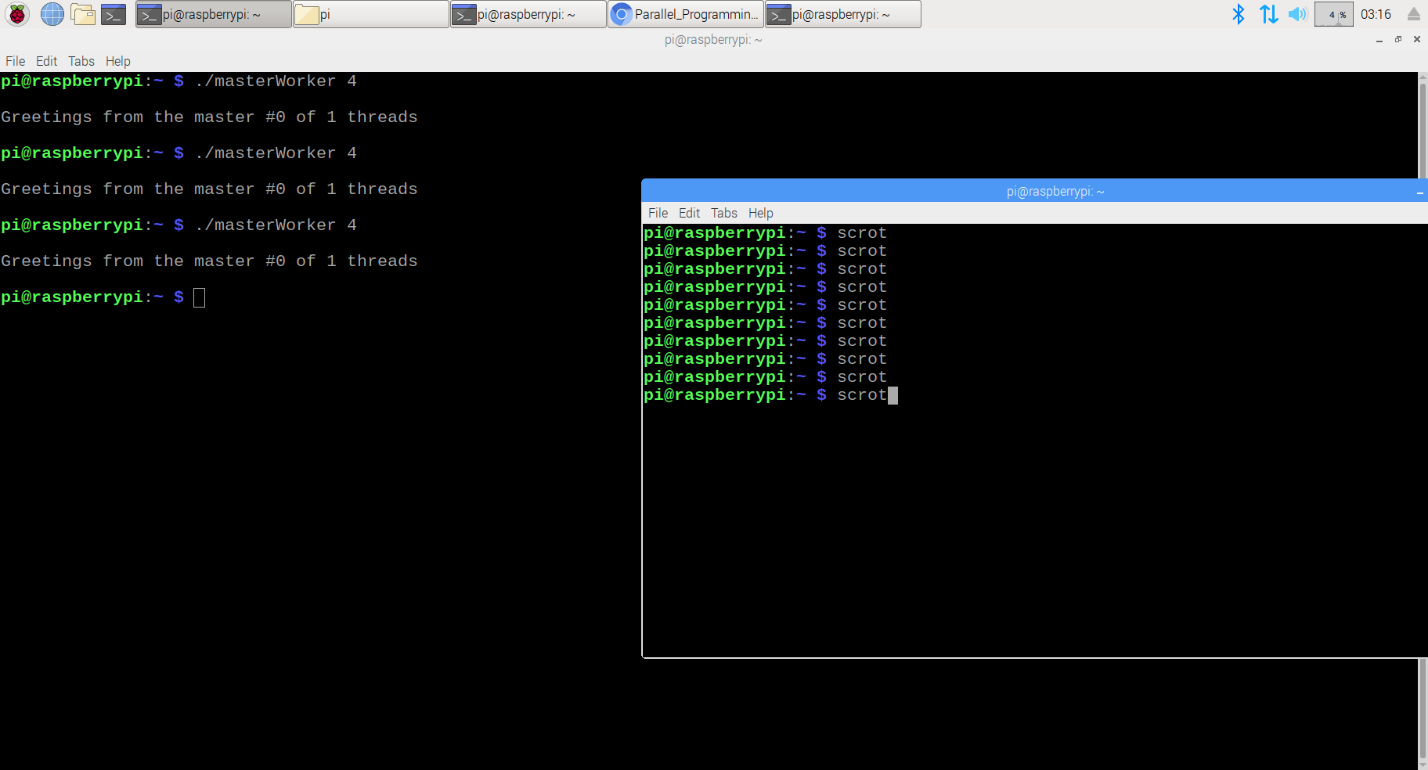


**Part Three: Program Structure: The Master-Worker Implementation Strategy**

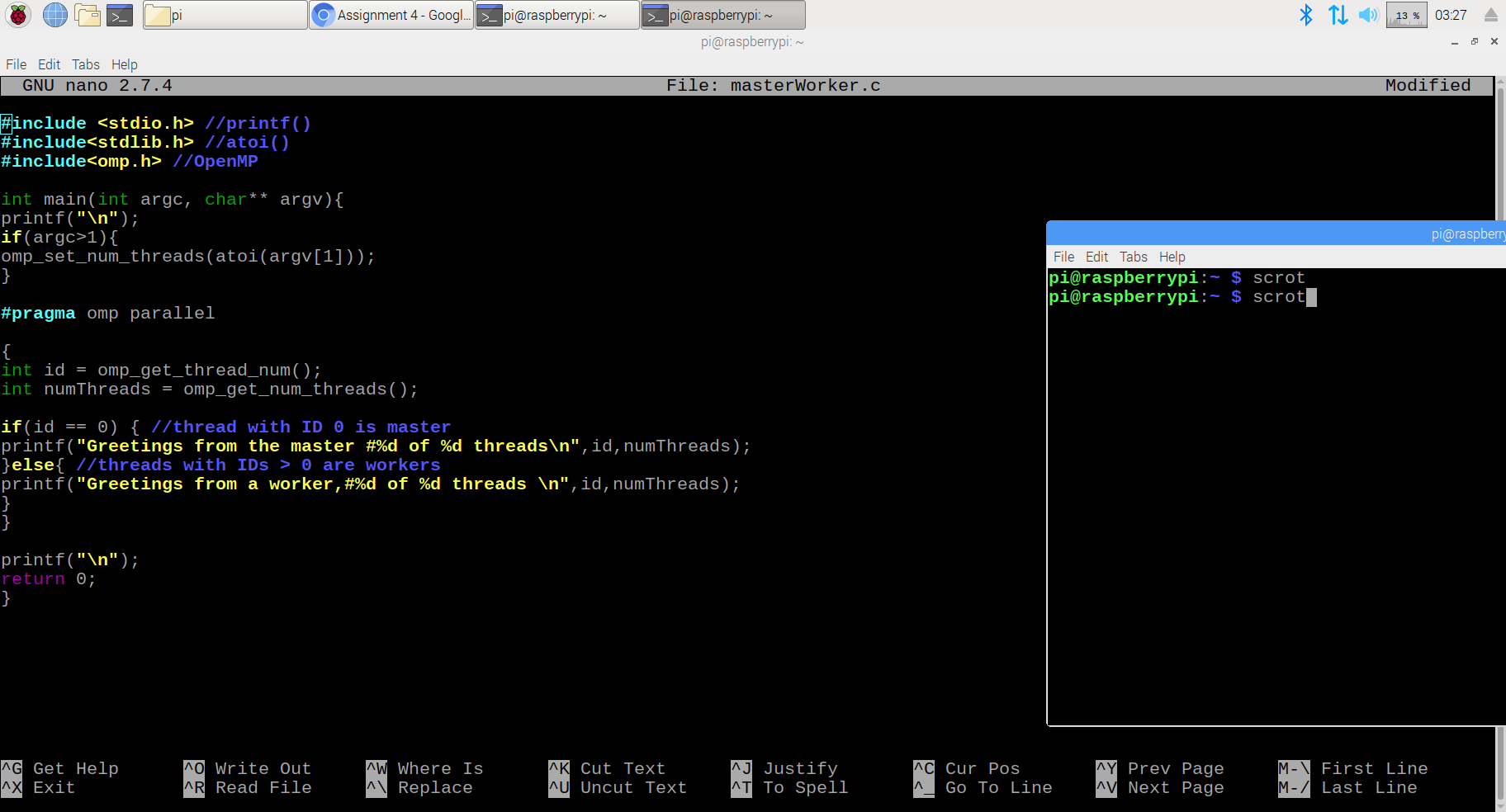
For this portion of the task, I created a file called masterWorker.c and once again copied the code from the provided pdf.



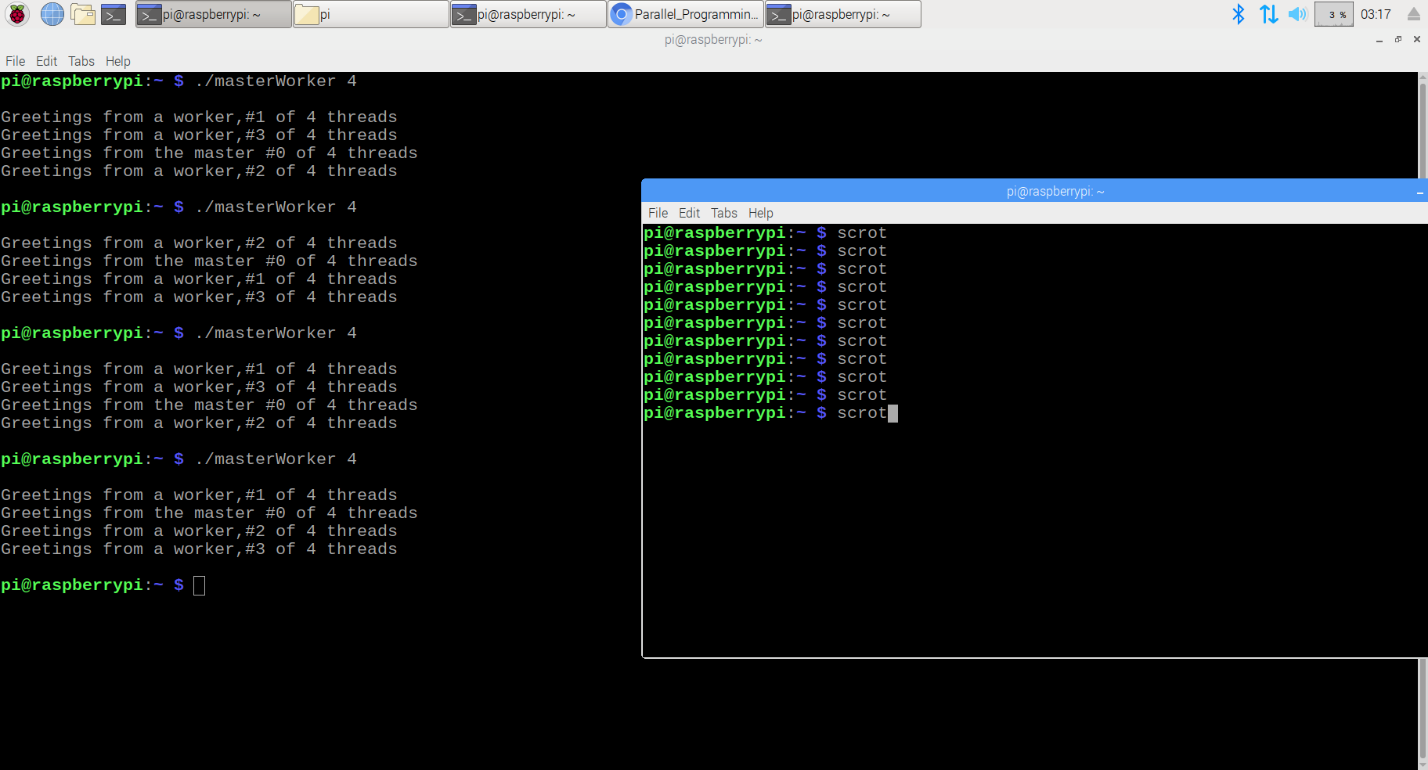
I created an executable to check the results of the program with the “#pragma omp parallel” line commented out.



As expected, only the first condition of the if statement was executed since it was not run parallel. I uncommented the “#pragma omp parallel” line and checked the results again.

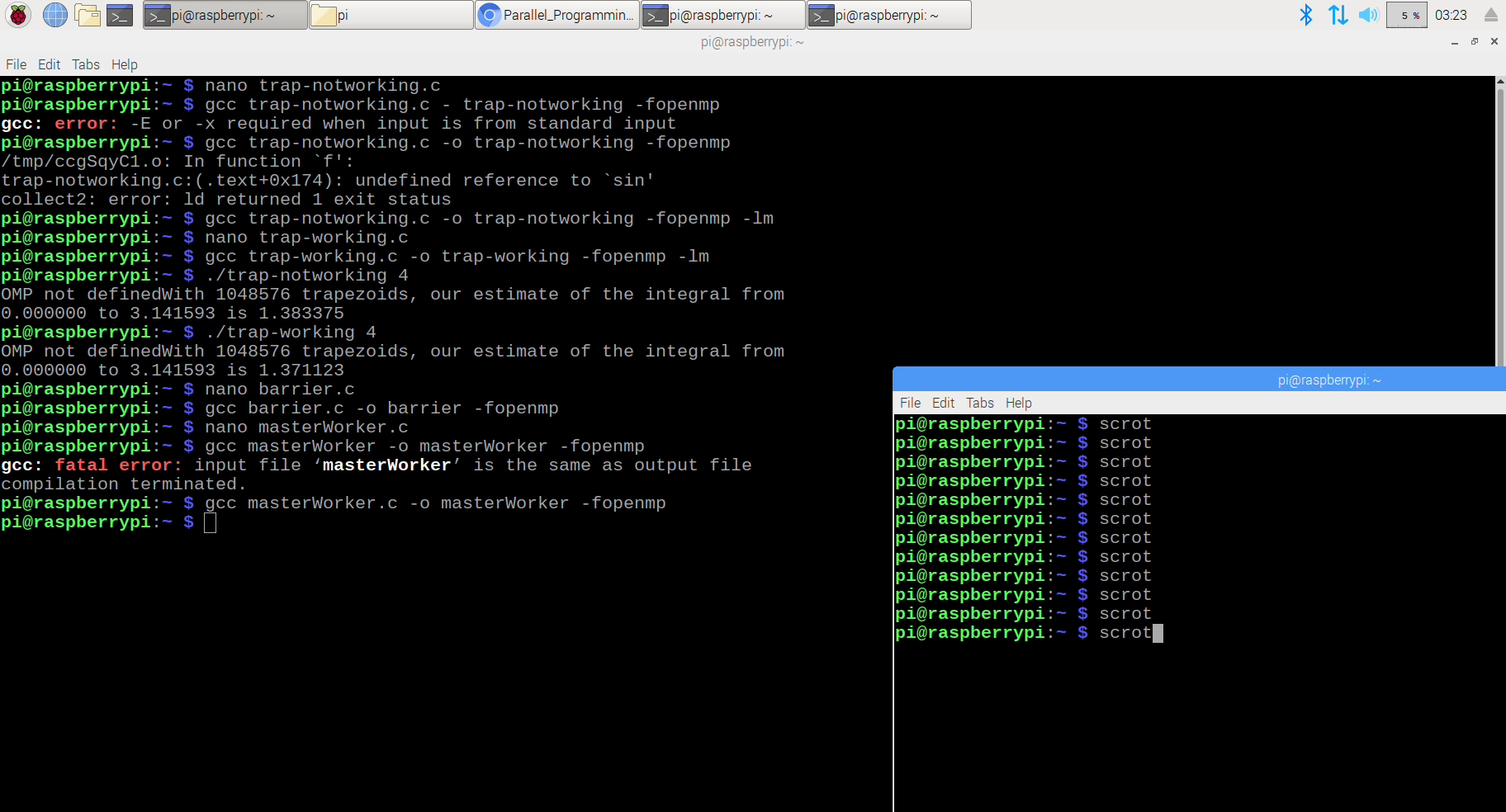


The line is now uncommented. The results shown are below.



Now we are able to see the workers and their corresponding thread number. There are only four threads because it is 4 core. Everything is as expected!

Below is an image of the creation of each of the files and their corresponding gcc line which created the executable for each file.



I had to add “-lm” at the end of the gcc lines for the trapezoidal program because it threw an error stating “undefined reference to sin”. That was the only major issue I ran into during the completion of this task.