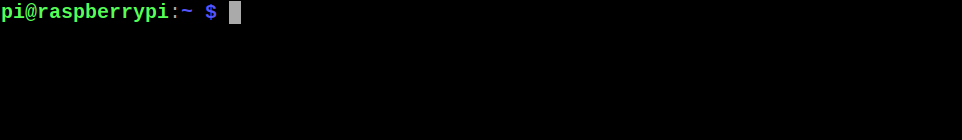
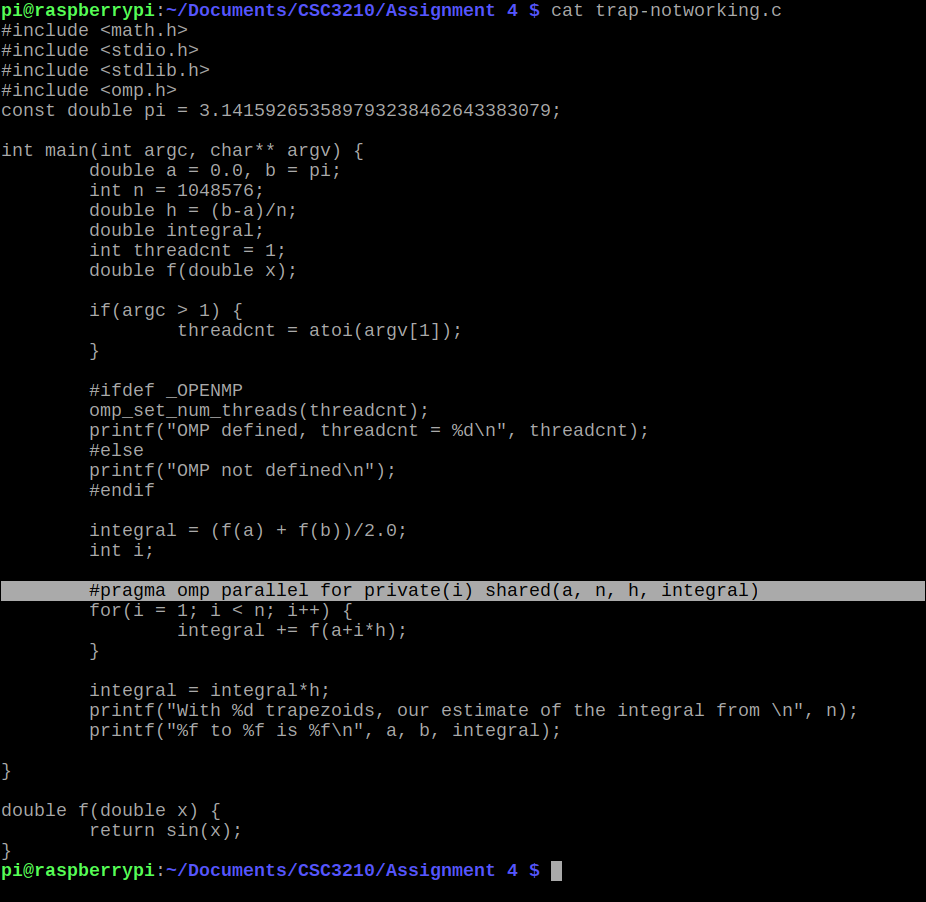
**Parallel Programming Task 3**

**1.1**

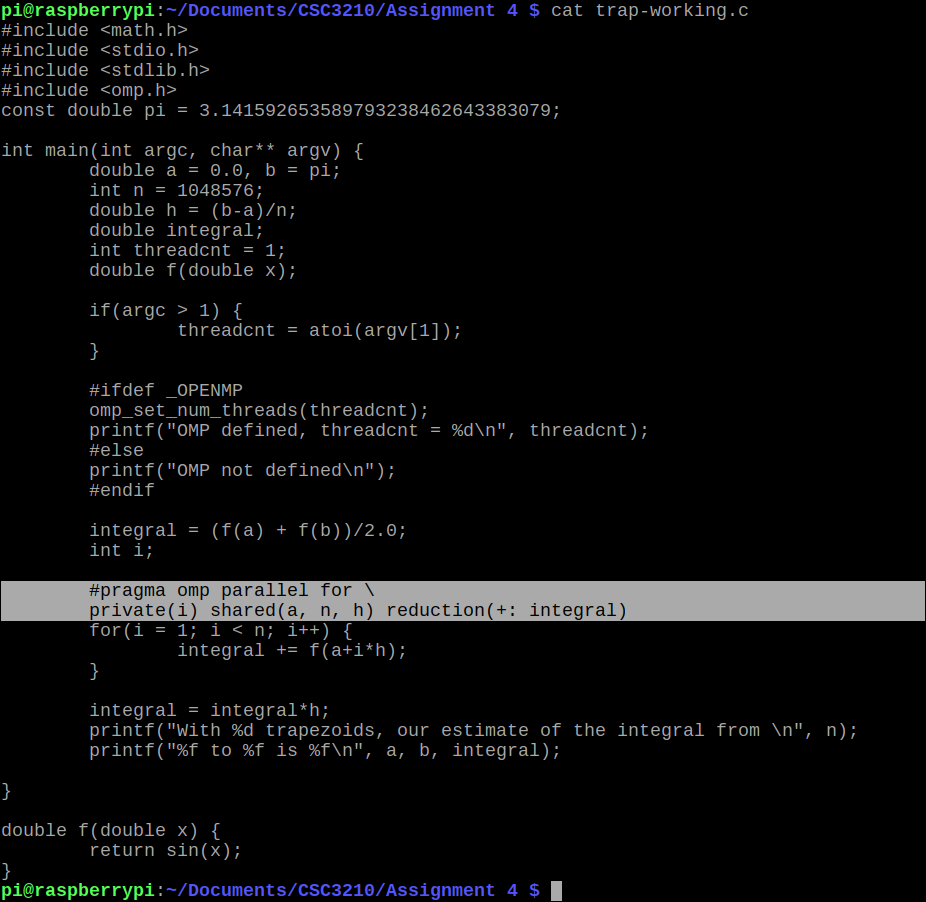
****

The above screenshot shows the terminal open in Raspbian on Raspberry Pi B.

**2.1**

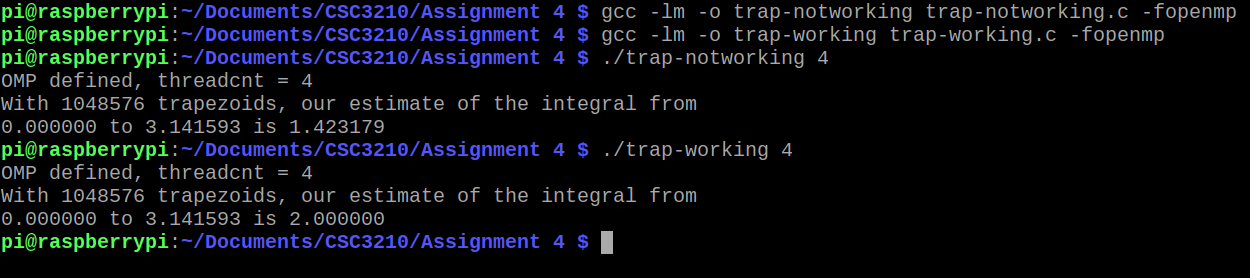
****

This screenshot shows a program, “trap-notworking.c”. Its intended function is to approximate the integral of sin(x) from 0 to π using a trapezoidal Riemann sum with 2^20 subdivisions. However, there is an error. Line 30 (highlighted above) declares the #pragma omp parallel directive with variable ‘i’ as private and ‘a’, ‘n’, ‘h’, and ‘integral’ as public. However, declaring the variable ‘integral’ as public is not sufficient for it to be properly summed across the parallel threads. Instead, this should be changed to add a reduction(+: integral) clause, as shown below.

****

The above screenshot shows a modified version of the program. In this modified program, “trap-working.c”, the #pragma directive is changed so that a reduction clause is added for ‘integral’.

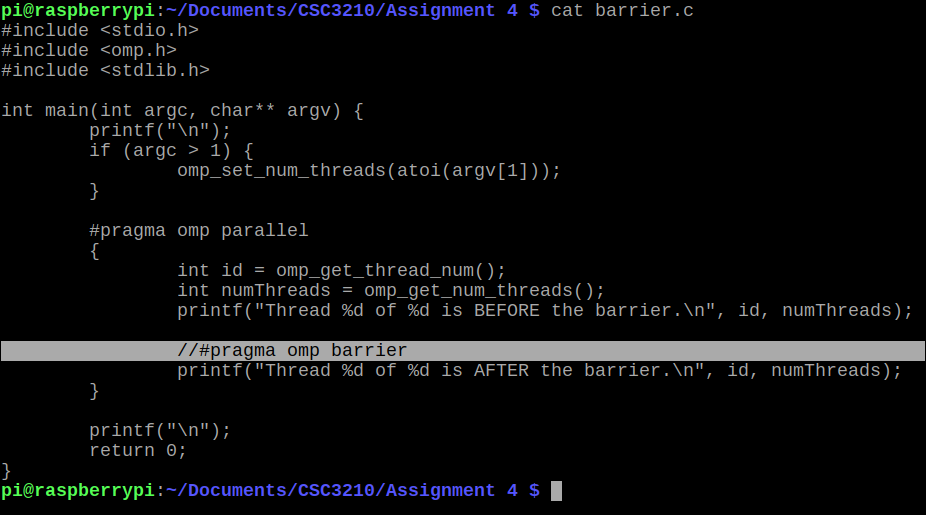
**2.2.**

****

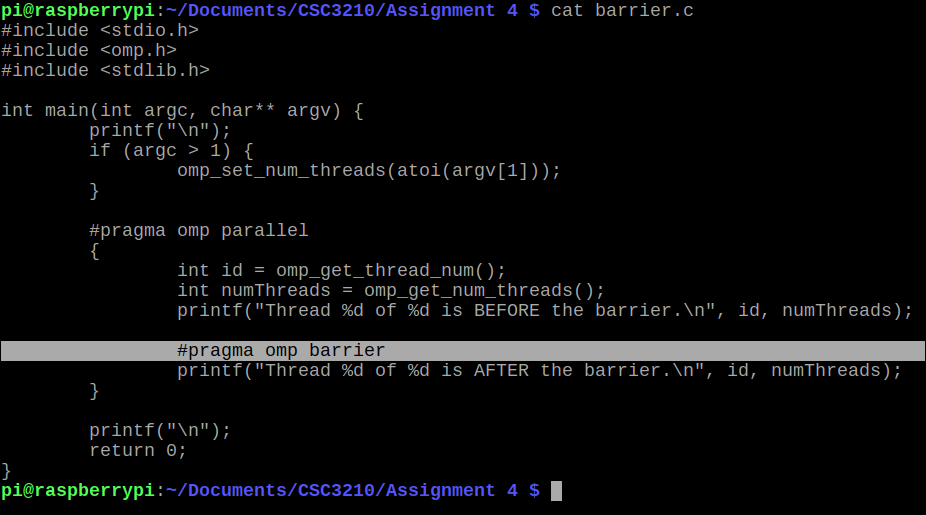
The above screenshot shows tests of the two programs, trap-notworking, and trap-working.

By antidifferentiation, we know that:

Thus, the exact integral is 2, and the expected approximation should be 2.0. However, as you can see, the two programs give different estimations for this integral. The first program gives a low estimate because it does not properly sum the ‘integral’ variable in parallel, but the corrected program gives the exact solution using reduction.

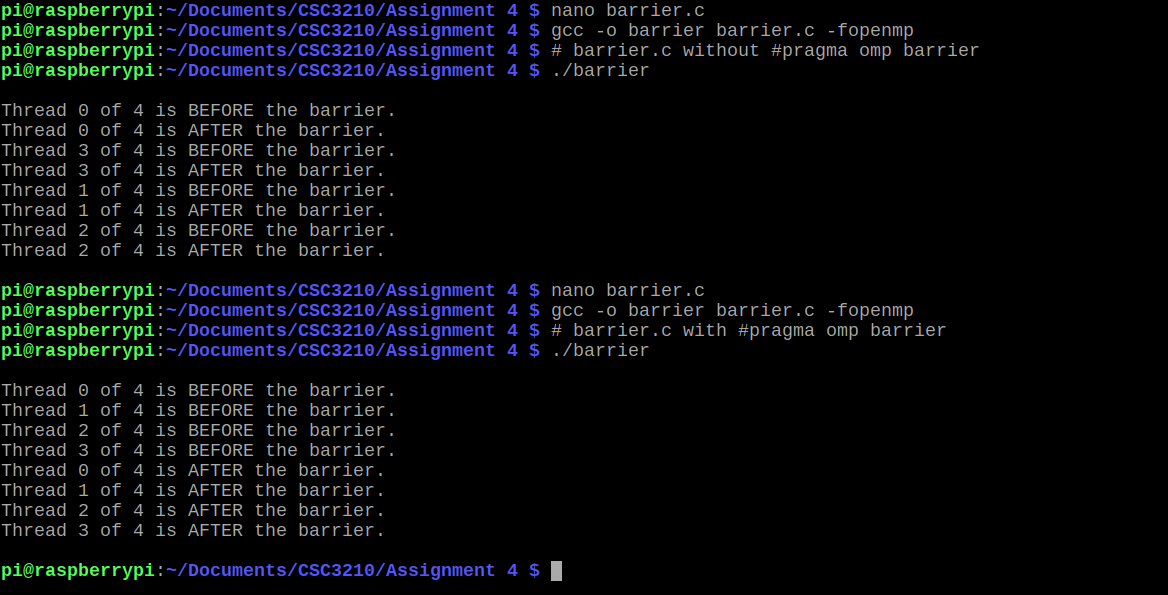
**3.**

The above screenshot shows another program, “barrier.c”. This is a simple program intended to demonstrate the use of barriers in parallel computing. For demonstration purposes, the “#pragma omp barrier directive” (highlighted above) that creates the barrier is commented out, and in the screenshot below it is again uncommented.

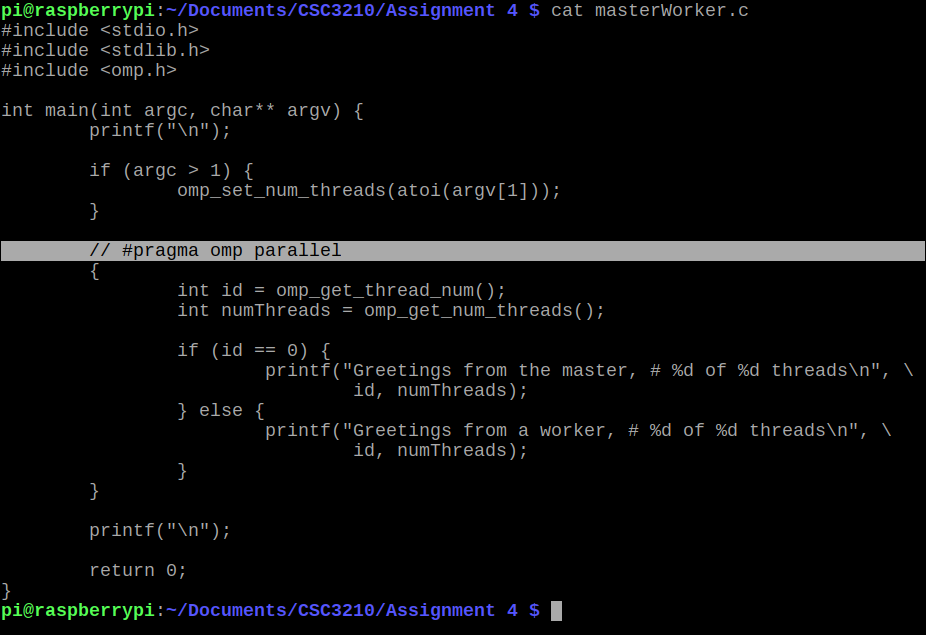
****

The screenshot below tests the barrier program both with and without the barrier directive, to illustrate the difference.

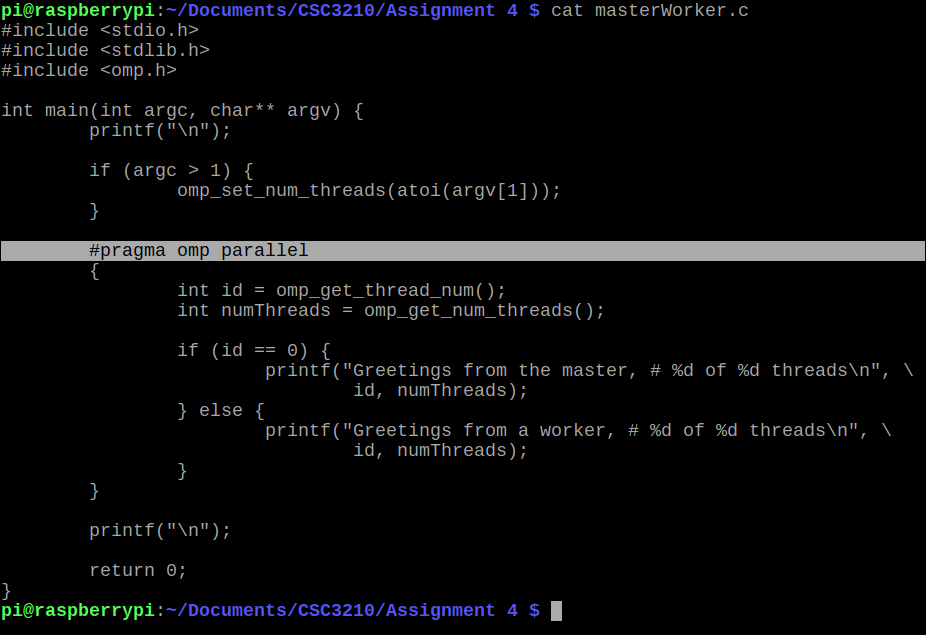
As you can see, in the first test, without using the “#pragma omp barrier directive”, each of the threads reaches and passes the barrier point before the next thread. However, when the “#pragma omp barrier” directive is added back in, all threads arrive at the barrier point and wait until the other threads have also reached before proceeding to the next section. This is, in essence, the function of barriers in parallel programming.

****

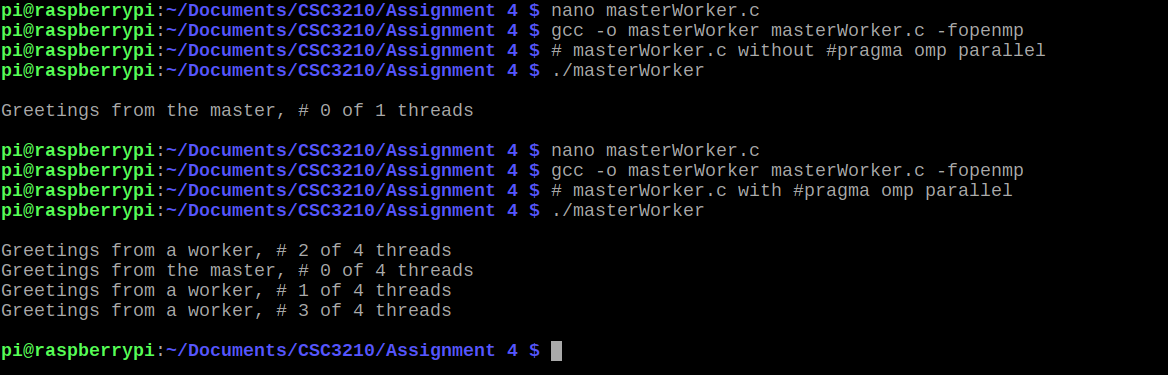
**4.**

****

The above screenshot shows another program, “masterWorker.c”. This is another simple program intended to demonstrate the use of “Master Worker” structure implementation in parallel programming. For demonstration purposes, the “#pragma omp parallel” directive (highlighted above) is commented out. In the screenshot below, it is again uncommented.

****

The final screenshot shown below tests this program, both with and without the “#pragma omp parallel” directive. As you can see, the sequential version of the program prints only one statement, while the parallel form prints four (in each case, one for every running thread). The role of “Master” in this case is tied to the thread id of 0, and all other threads are workers (So even when there is only one thread – sequential – there is still a master).

****