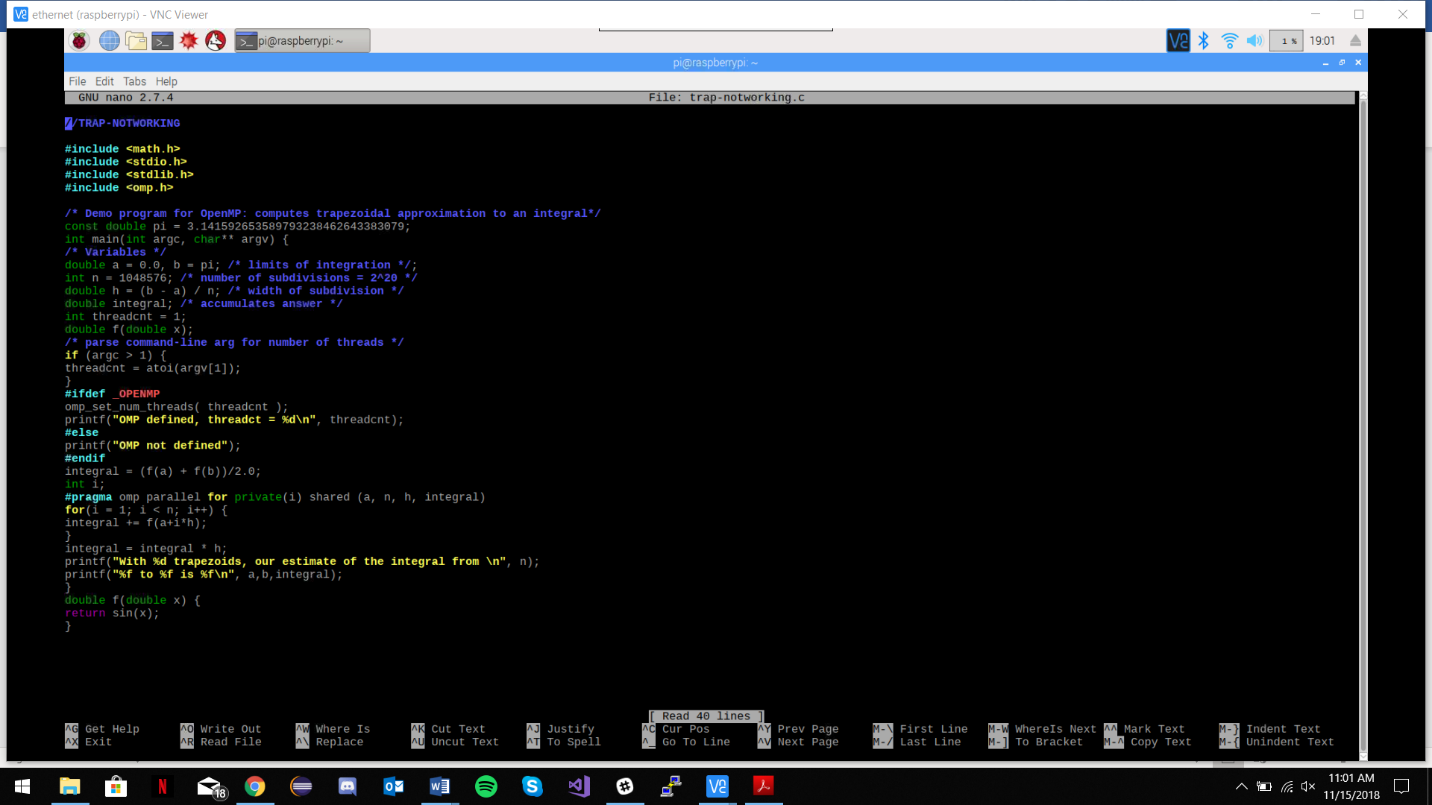
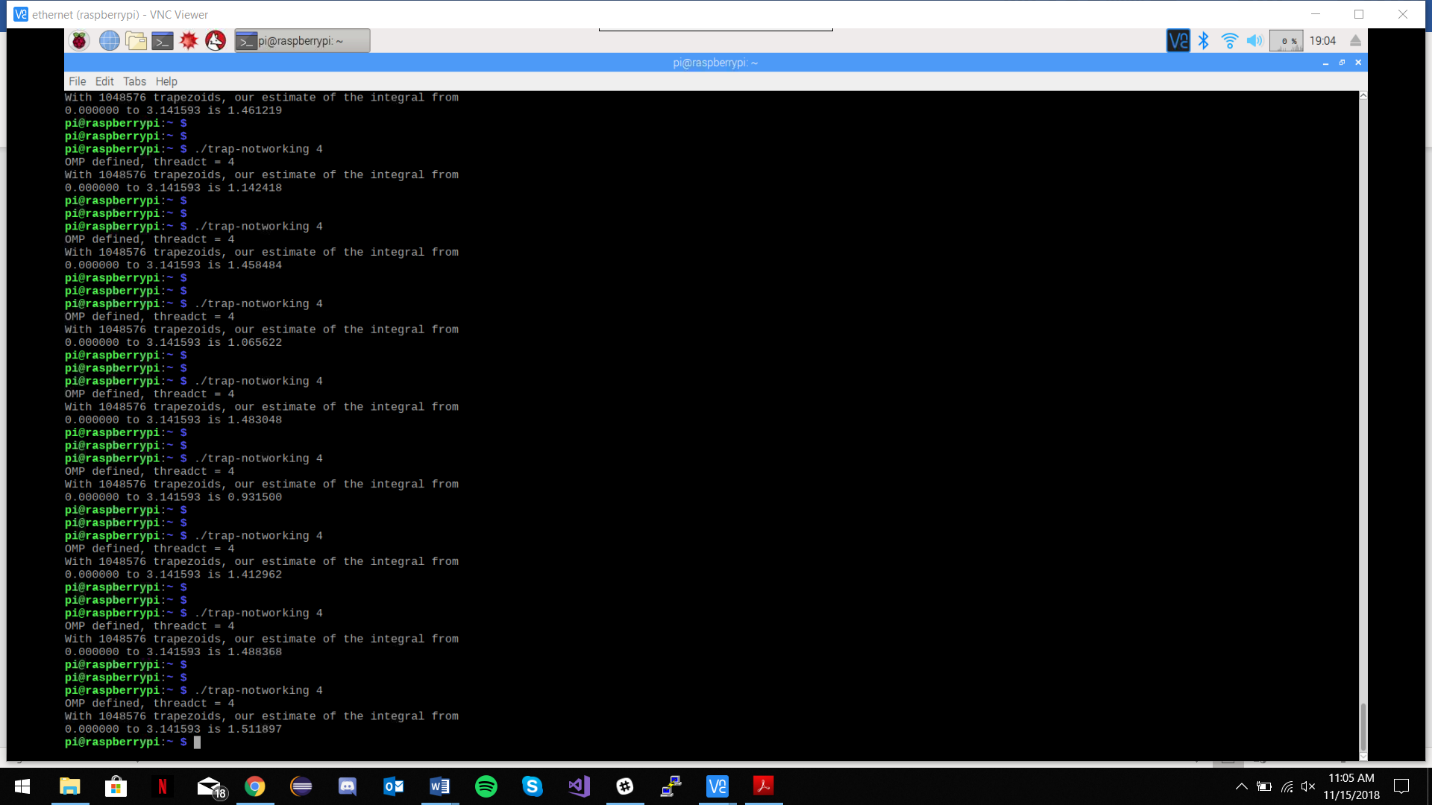
Trap-Notworking.c

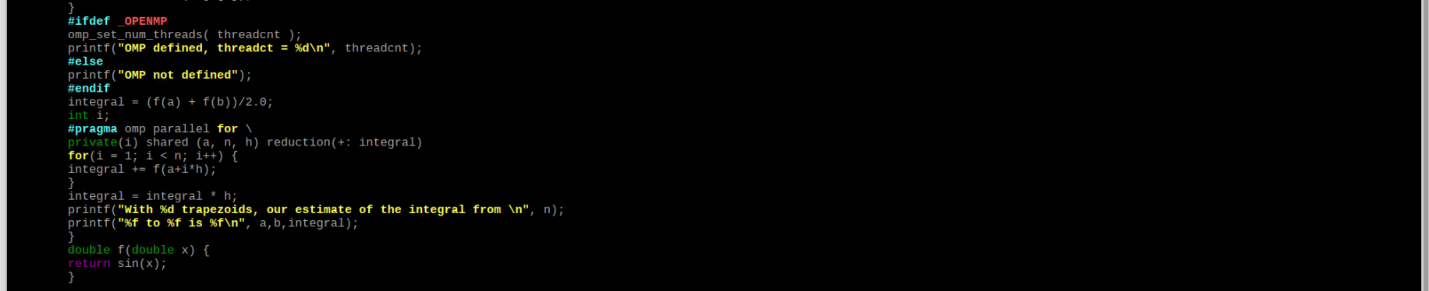


Had to implement -lm (after -fopenmp) to enable compiling.

Trap-Notworking results

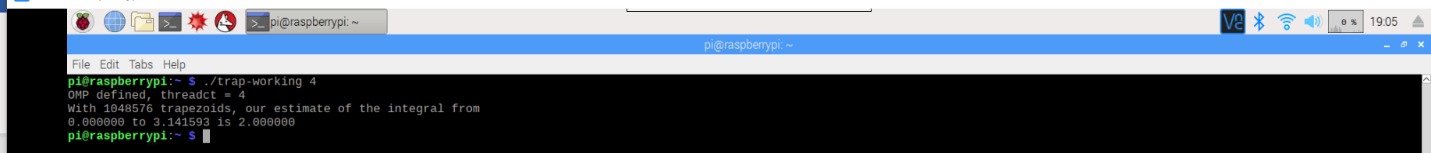
Since the Pi has 4 cores, it makes sense to set the thread number to 4. For this piece, the overall integral gave values varying from 1.2x to 1.5x. This contradicts the expected value of 2.0 and is not a consistent result. This is caused because each thread is not contributing correctly to the entire integral sum; a reduction statement will need to be implemented.

Trap-Working.c

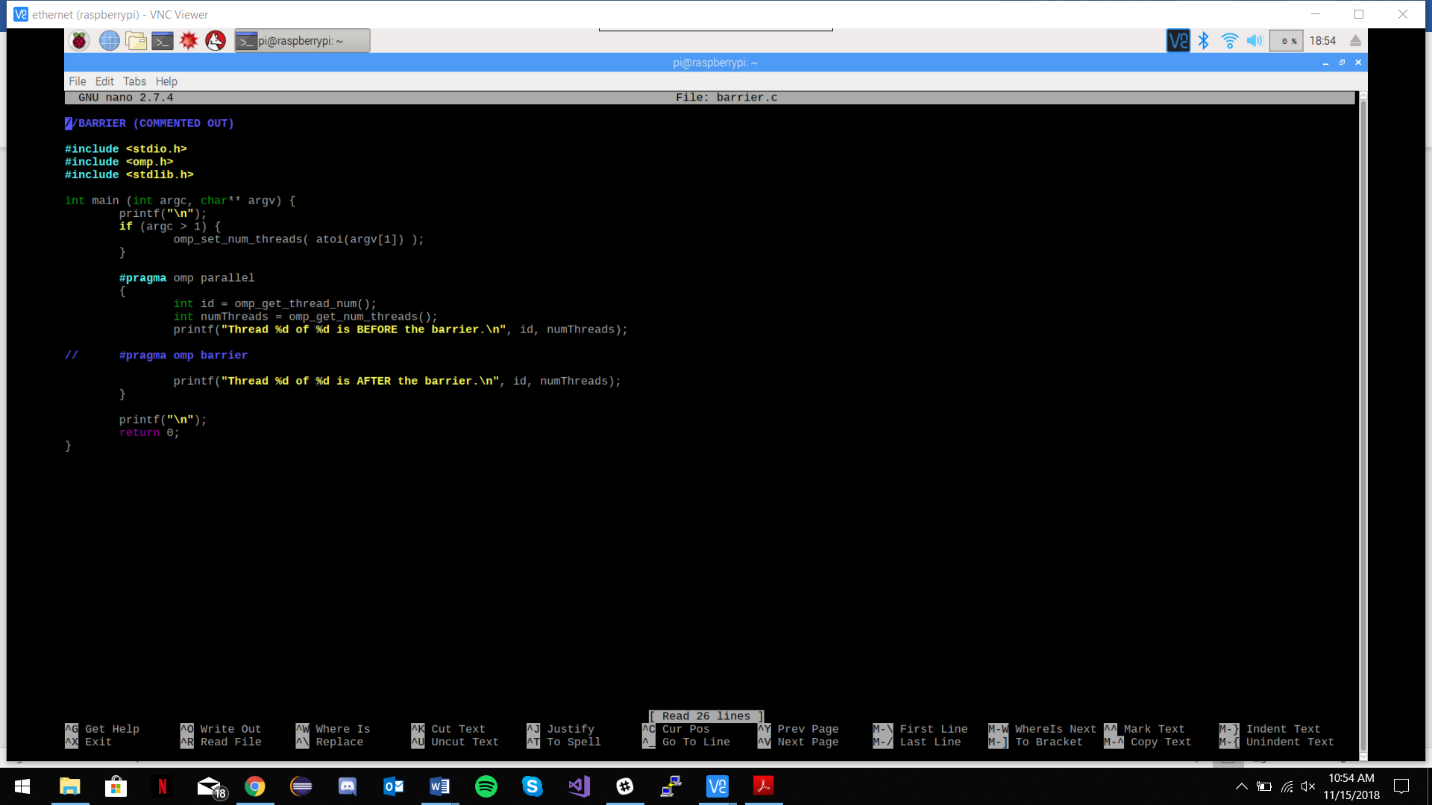


With the implementation of the “reduction(+:integral)” directive which allows all threads to correctly contribute to the overall integral sum, we now get the expected integral value of 2.0.

Trap-Working result

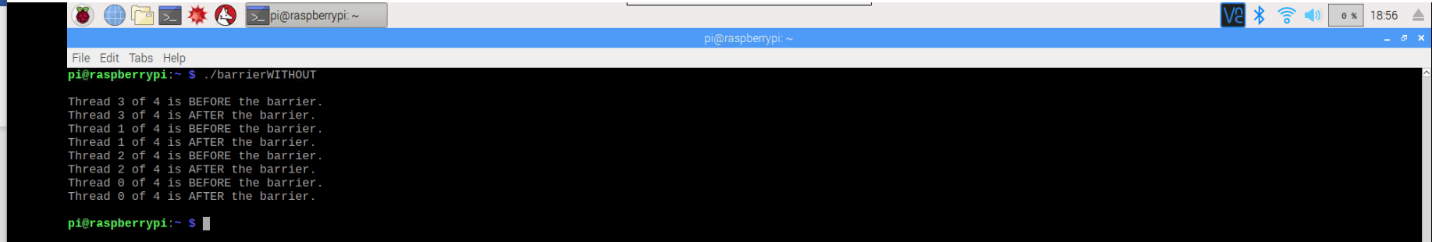


Barrier (“#pragma omp barrier” commented out)



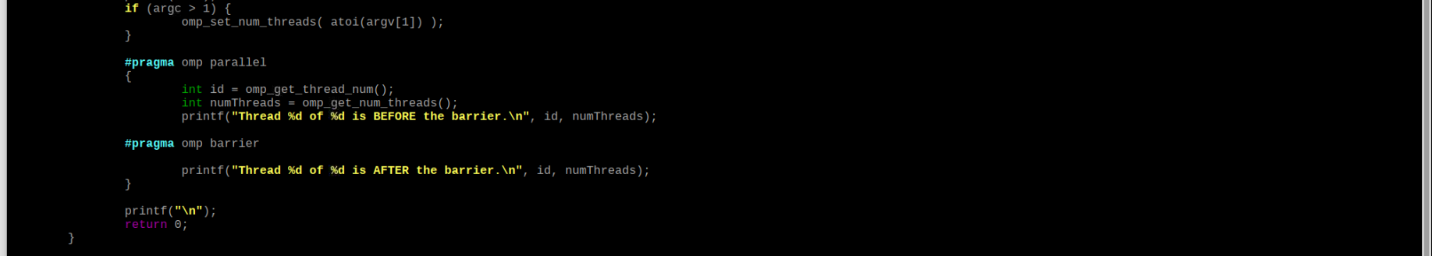
With the barrier line commented out, the entire #pragma omp parallel segment runs for each thread, all the way through, one at a time.

Barrier (“#pragma omp barrier” commented out) result



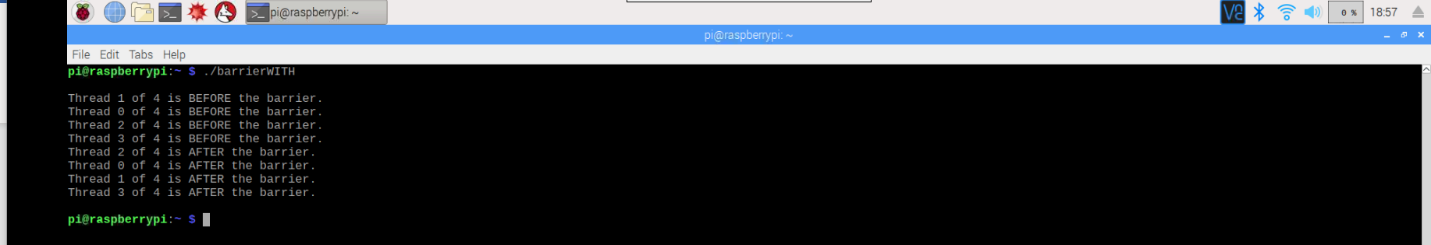
In this code, each thread runs through the entire block of printf statements before handing over control to the next thread. It is important to note that the order of the threads is random.

Barrier (“#pragma omp barrier” active)



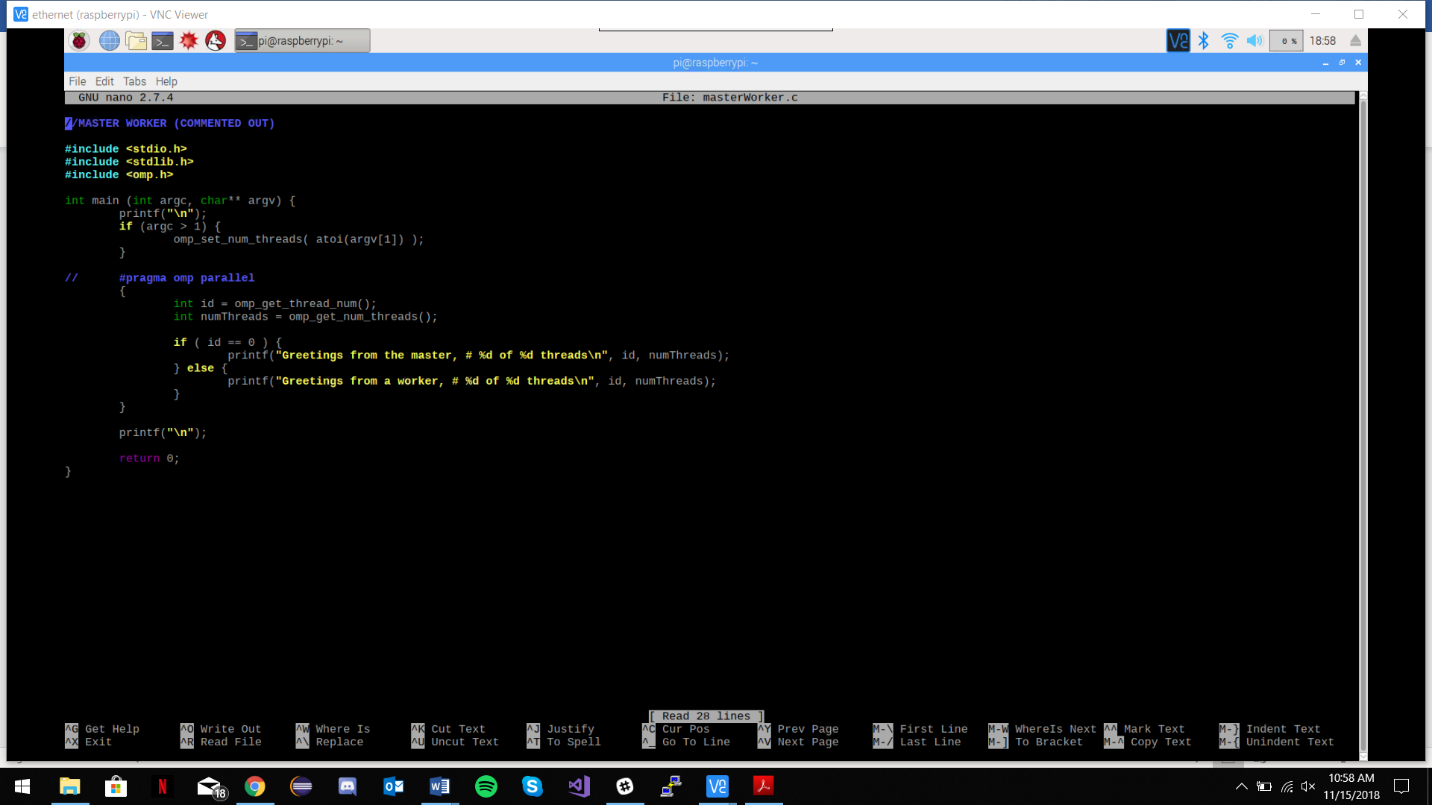
Now that the barrier control statement is active, it should act as a sort of waiting area where each thread will run and if it isn’t the last thread to run the first part of the code, it will wait until all other threads have run to that point.

Barrier (“#pragma omp barrier” active) result



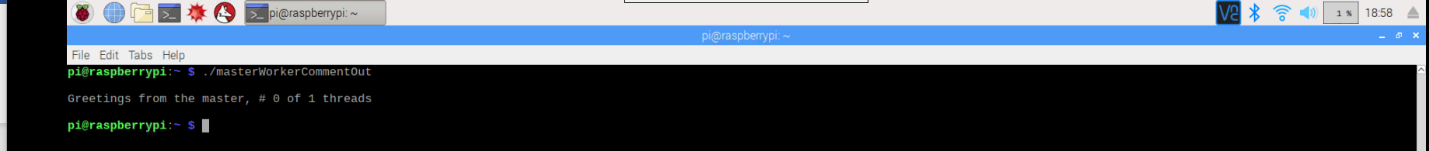
Here, with the barrier control statement, each thread runs the first printf statement before the barrier statement and then waits for the rest of the threads to run the same printf before moving on. It is interesting to see here that the thread order is “reset” after the barrier statement. The threads after the barrier statement don’t necessarily run in the same order that they did before the barrier statement. This shows that there is no queue or memory of the initial order keeping the threads in the same order.

Master-Worker (“#pragma omp parallel” commented out)



With the parallel control statement commented out, it is expected that only one thread will run, and it will be treated as #0, which should activate the “if” statement and print the “master” printf statement.

Master-Worker (“#pragma omp parallel” commented out) result



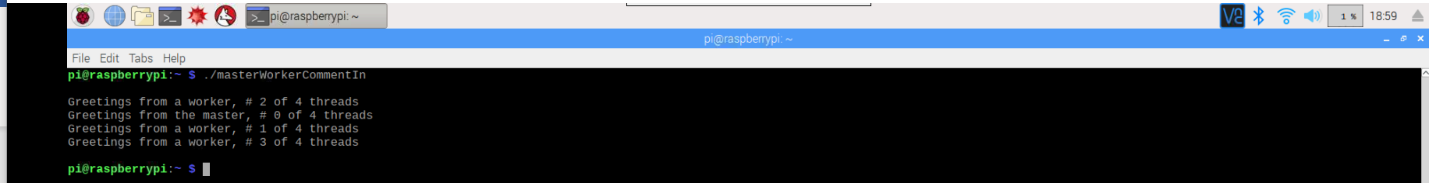
Since the parallel declaration is commented out here, then only one thread runs, and is considered to be the first thread in 1, and therefore used the “if” statement since this thread is treated as thread 0. It is interesting to see that, seemingly, the logical #1 core is the one always used when the cpu isn’t being treated as parallel. Another possiblity is that the core being used is random, but is being treated as the only core and therefore is assigned as thread #0.

Master-Worker (“#pragma omp parallel” active)



With the parallel declaration active, we should see thread #0 become the master, and threads #1-#3 be assigned as workers. This is because the program will now use all 4 cores of the Pi’s cpu.

Master-Worker (“#pragma omp parallel” active) result

 Now that the parallel control is enabled, each thread runs and we have a seperation between the “master” thread and the “worker threads”. This implementation can be very useful to ensure that certain cores are assigned specific tasks that the programmer wants those cores to handle.