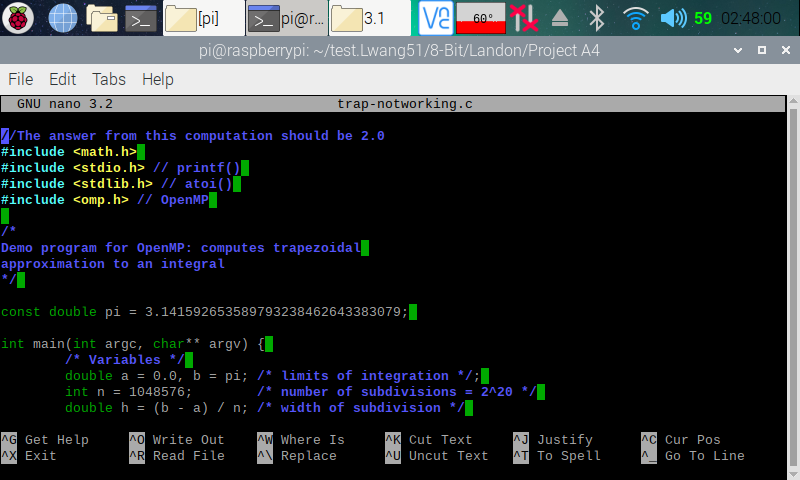
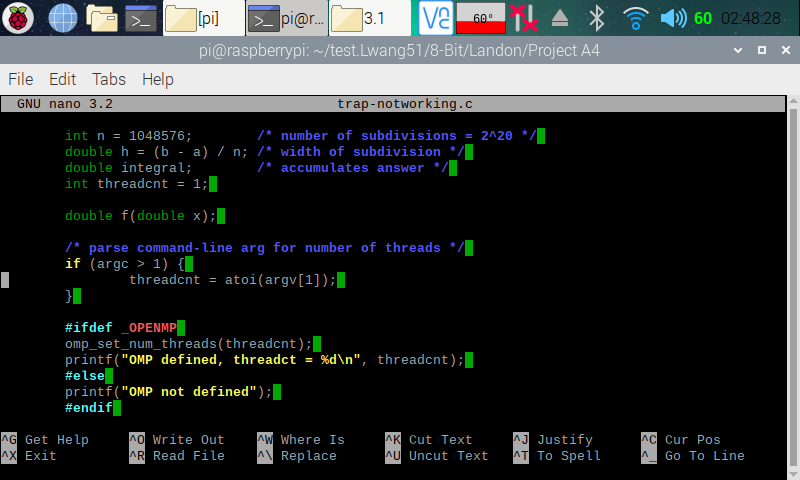
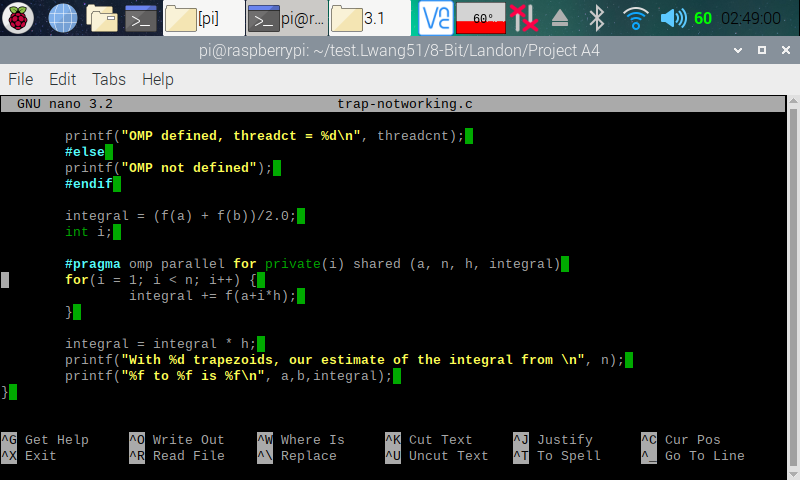
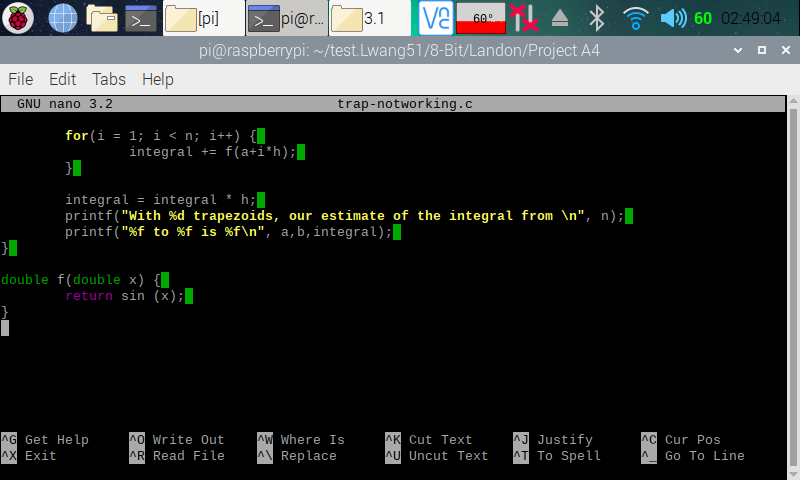
**Parallel Programming Basics**

**Integration Using the Trapezoidal Rule**

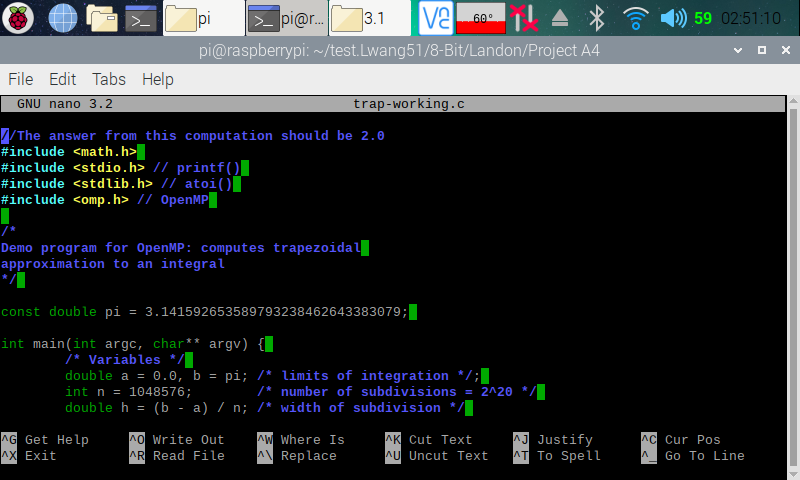
****

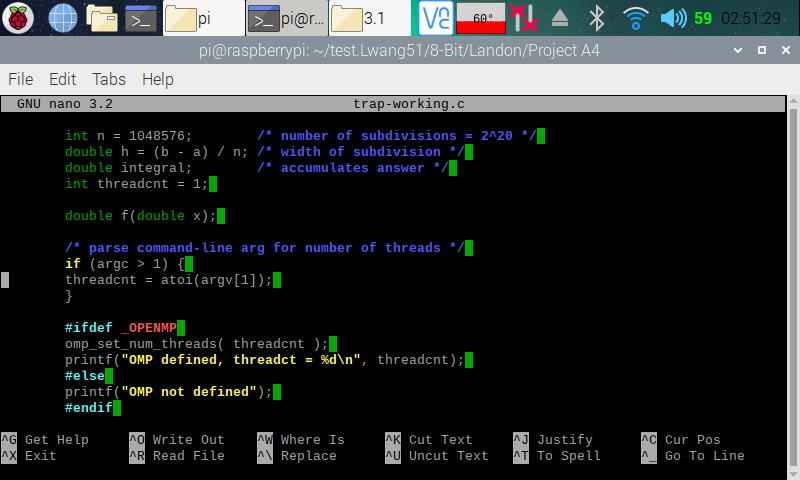
****

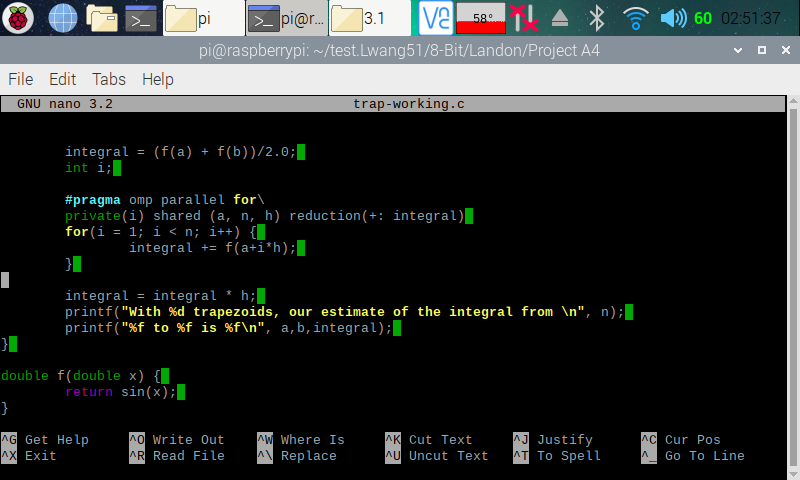
****

****

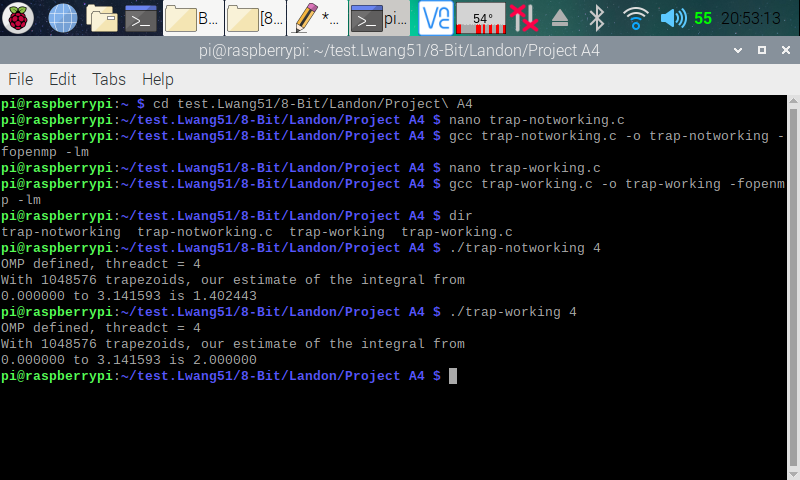
Here (in the four screenshots above), I copied and pasted the codes from the Parallel Programming Task A4 document and used the nano editor to create a program on my Raspberry PI.

****

****

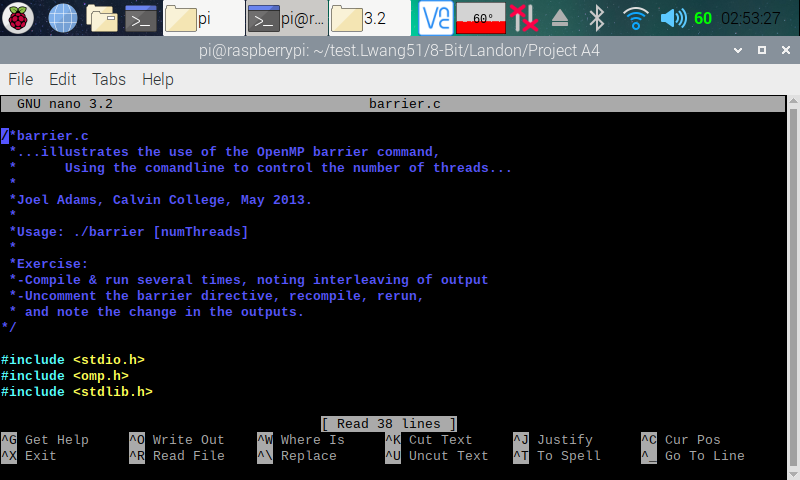
****

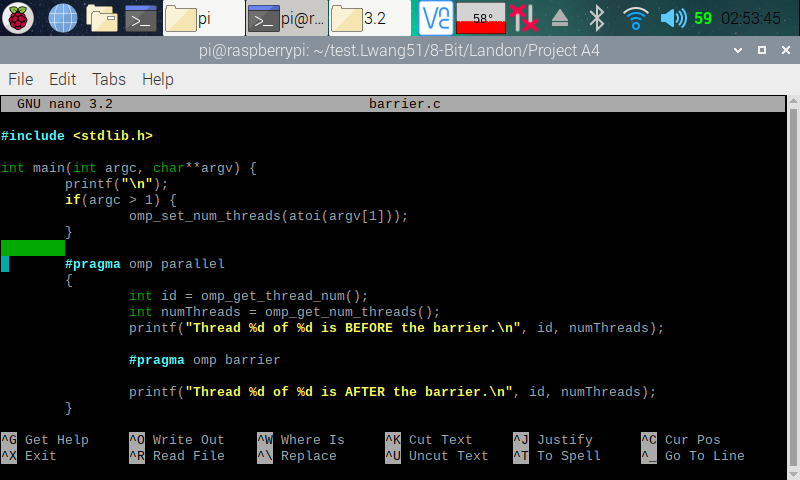
Here (in the three screenshots above), I copied and pasted the codes from the Parallel Programming Task A4 document and used the nano editor to create a program on my Raspberry PI.

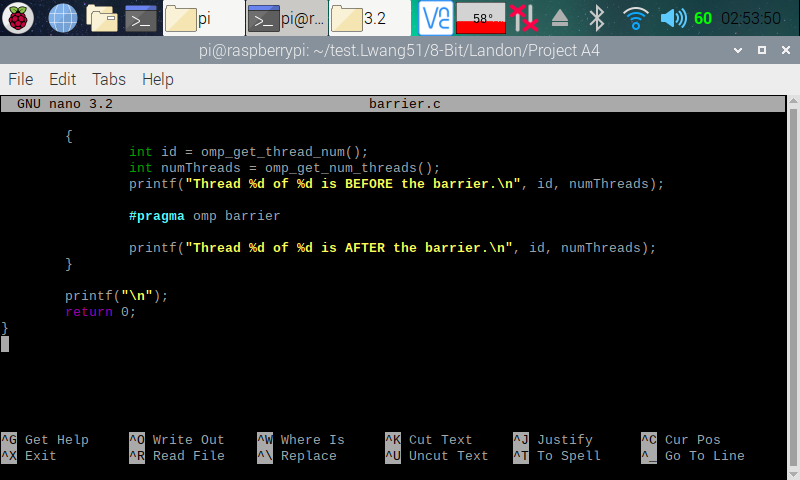
****

Here (in the screenshot above), I made the executable program for both the trap-notworking.c and trap-working.c programs. I then ran both of the program with a command-line argument that specified to have four threads to b forked. The trap-notworking.c program displayed the result of 1.402443, which we know is not correct from the program comments. The trap-working.c program displayed the result of 2, which we know is correct from the program comments. The reason that trap-notworking.c program displayed the wrong result was because the variable ‘integral’ is being shared across the forked threads. To fix this, we declared the variable ‘integral’ as a reduction pattern, as shown in trap-working.c program.

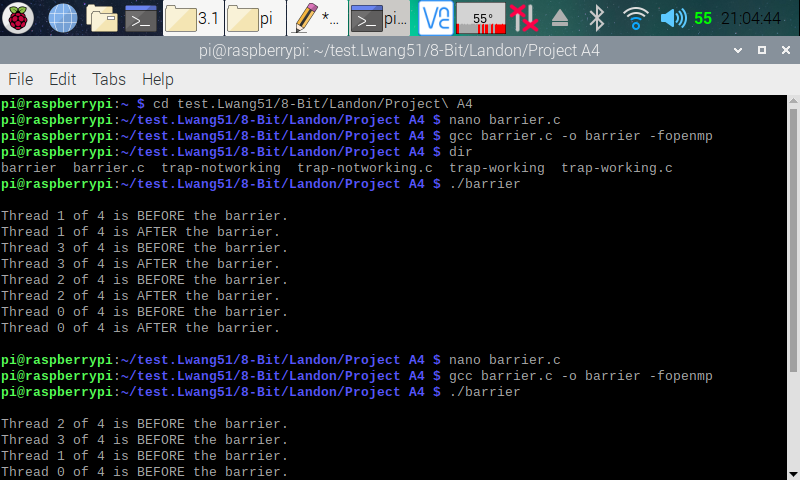
**Coordination: Synchronization with a Barrier**

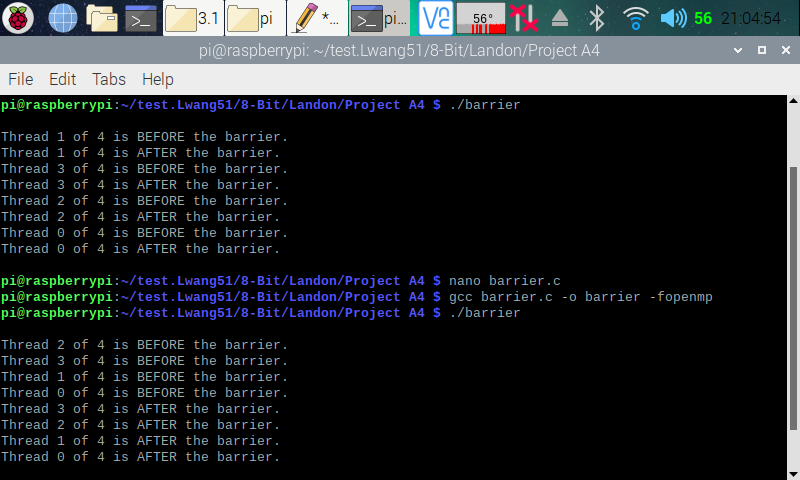
****

****

****

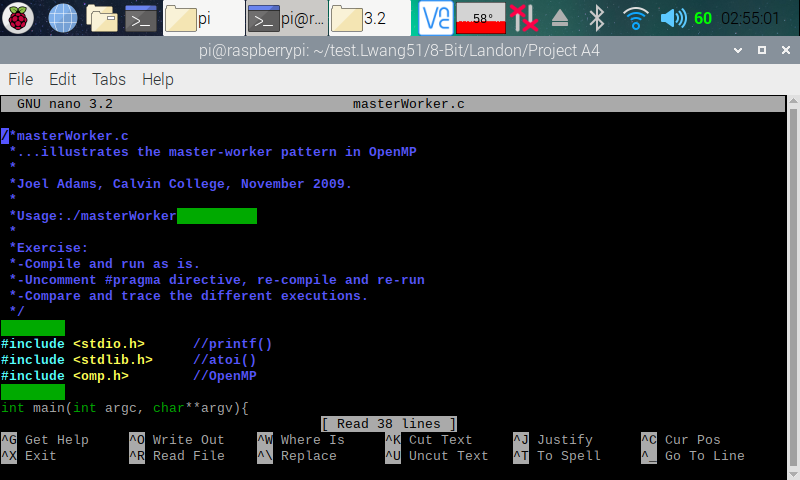
Here (in the three screenshots above), I copied and pasted the codes from the Parallel Programming Task A4 document and used the nano editor to create a program on my Raspberry PI.

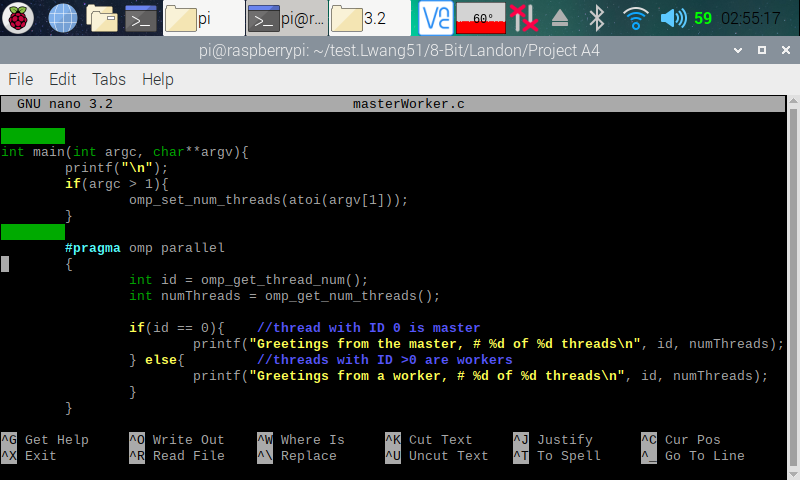
****

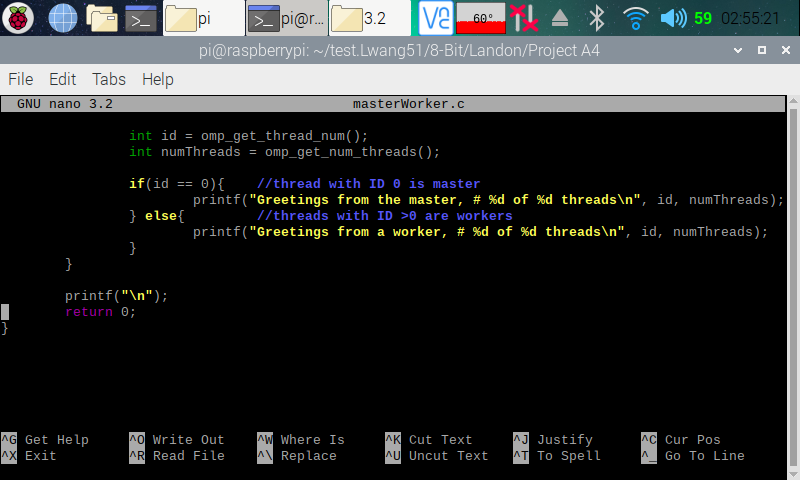
****

Here (in the two screenshots above), I made an executable file for the barrier.c program. I ran it one time (the first run) with line 31 (#pragma omp barrier) commented out and another time (the second run) with line 31 uncommented. In the first run, we can see that the threads went on and executed their next statement (Thread \_ of \_ is AFTER the barrier) without waiting for the other “before” threads to finish their execution. In the second run, with the barrier enforced, we see that all threads waited for other threads to finish their “before” threads executing the “after” threads. This is because the barrier forced all threads that reached the point first to wait until all other threads reach the point before allowing them to continue.

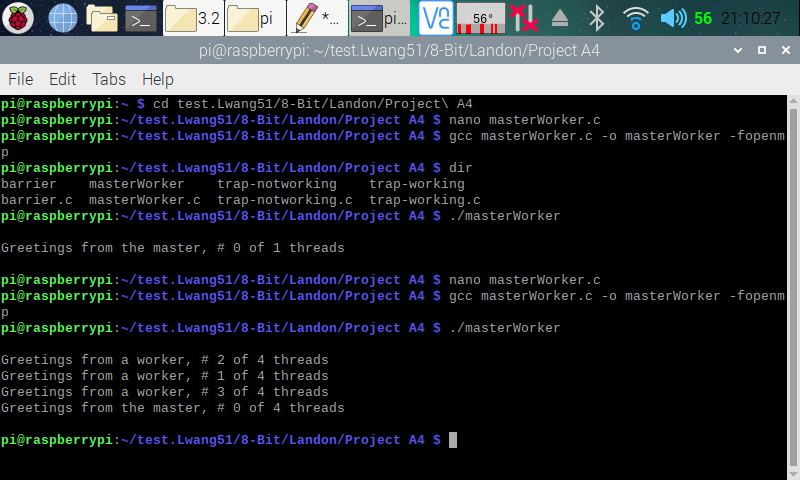
**Program Structure: The Master-Worker Implementation Strategy**

****

****

****

Here (in the three screenshots above), I copied and pasted the codes from the Parallel Programming Task A4 document and used the nano editor to create a program on my Raspberry PI.

****

Here (in the screenshot above), I created an executable file for the masterWorker.c program. I ran it one time (the first run) with line 24 (#pragma omp parallel) commented out and another time (the second run) with line 24 uncommented. In the first run, we can see that only a master was created. This is because with line 24 commented out, the program did not run in parallel. After uncommenting line 24, workers were created as the program runs in parallel.