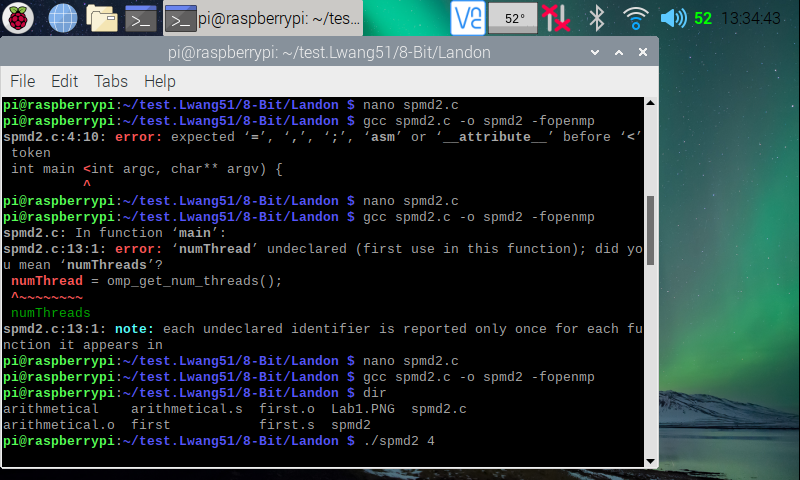
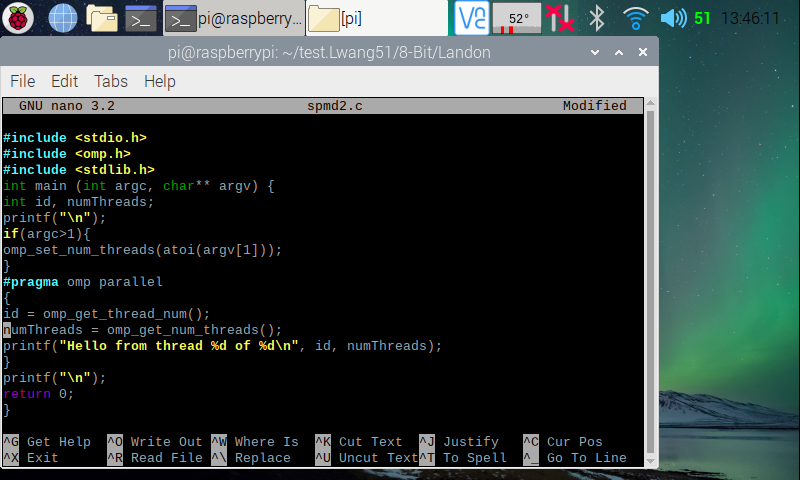
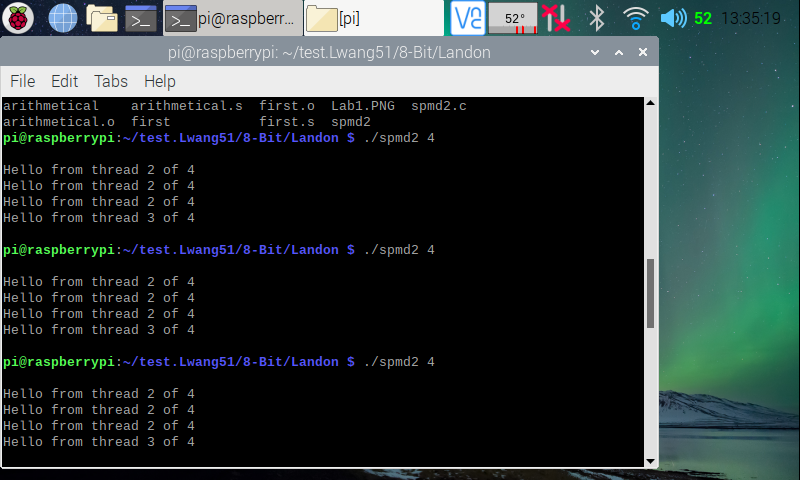
**GETTING STARTED WITH THE RASPBERRY PI AND PARALLEL PROGRAMMING**

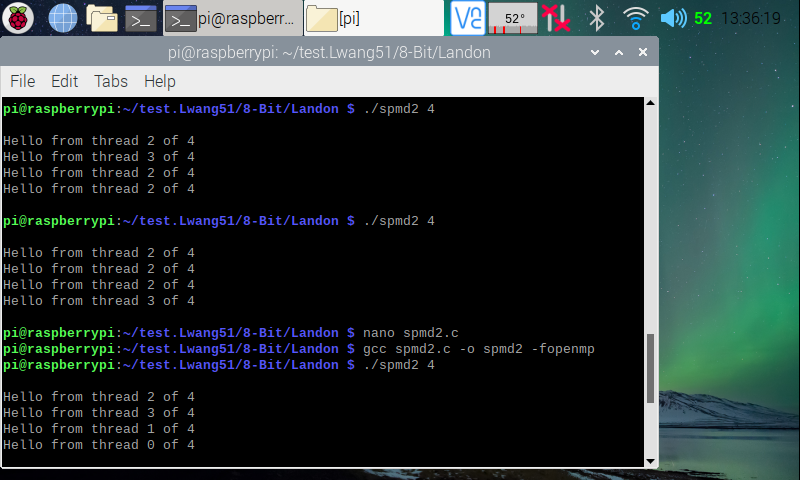


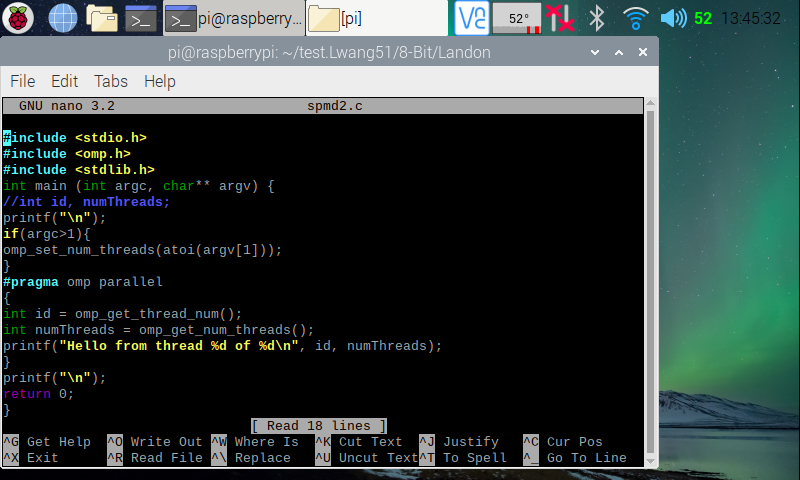


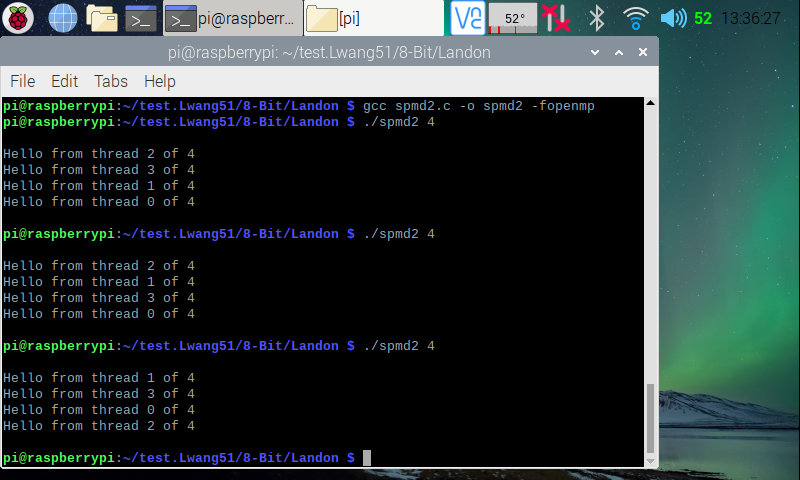
Here (in the two screenshots above), I created the spmd2 program using the instruction **nano spmd2.c** command. After writing/copying the program from the Getting Started with the Raspberry Pi and Parallel Programming file, I made an executable program file for it using the instruction **gcc spmd2.c -o spmd2 -fopenmp**. I ran into a few errors (typed a “<” instead of a “(” on line 4, 10th character and called “numThread” instead of “numThreads” on line 13) before getting the terminal to create an executable file for spmd2 program. After the executable file was created, I ran the program with 4 threads to fork using the instruction **./spmd2 4**.



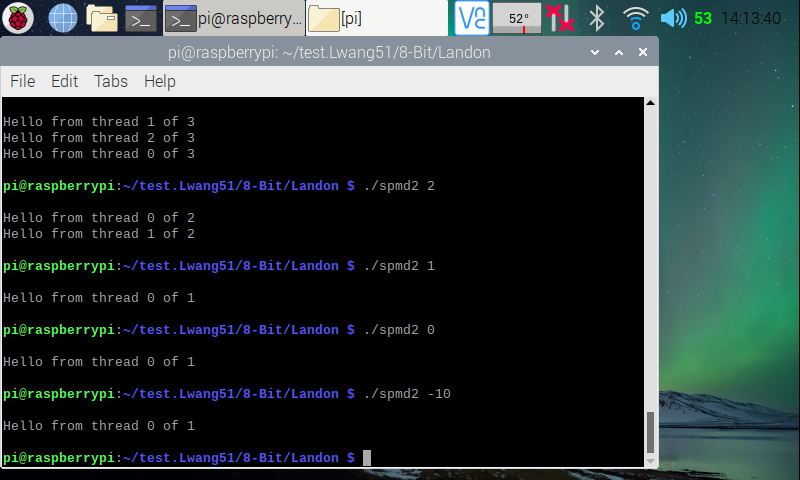
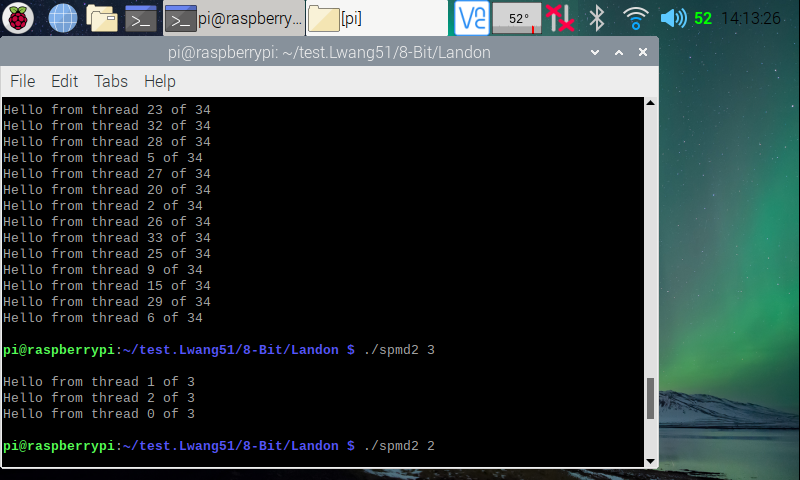
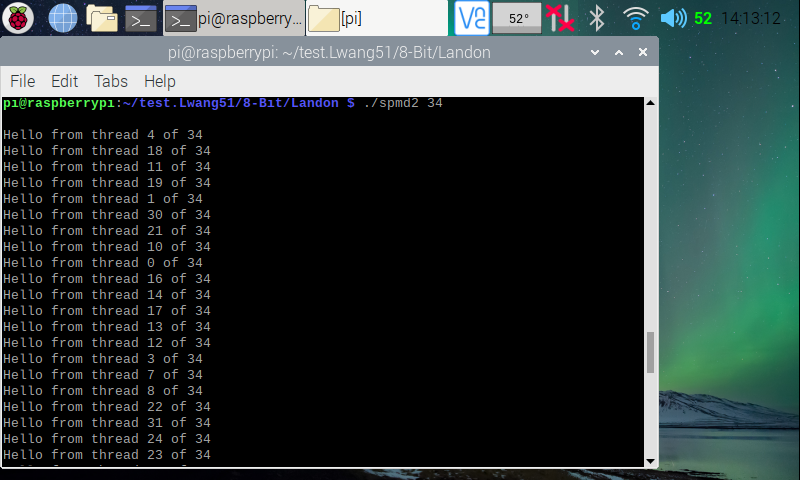
Here (in the screenshot above), I tested the spmd2 program a few times, each running with 4 threads to fork. From my observation, the program does run with four threads, but the thread id number 2 repeats many times. This is because, according to the Getting Started with the Raspberry Pi and Parallel Programming tutorial file, although the PI has multiple cores, the cores share the same memory bank in the machine. We declared variables outside of the “#pragma opm parallel” block, so all threads point to a same variable memory address.







Here (in the three screenshots above), I ran two more trails of the spmd2 program with 4 threads. Looking at the top one, we can see that the order in which the threads finished are different. This is natural, because it is never guaranteed which thread will finish first. To fix the id error, I went back in edited the code, then made a new executable file for it. In the code, I commented out “int id, numThreads” code on line 5, then made id and numThreads inside the “#pragma opm parallel” to full variable declarations. This way, when a new thread is formed, they each will have their own private copy of the two variables. I ran it again with four threads, and now, we can see that each thread has its own special id starting from zero. I then ran the program two more times, and we can see that each time it runs, the threads finish in different order.



Here (in the three screenshots above), I ran the program with different amount of threads (34; 3; 2; 1; 0; -10 threads). From my observation, running the program with n amount (greater than or equal to 1) of threads will create n amount of threads each finished in different orders. I tried with 0 threads and -10 thread (expecting an error), but it just executed with one threads. I guess that is the default.