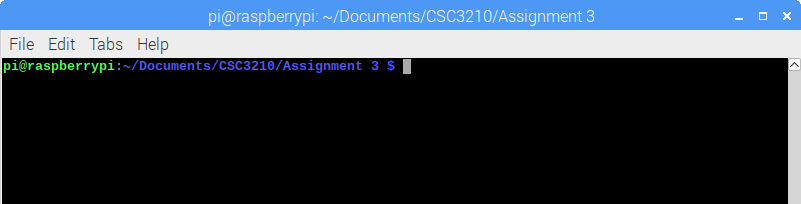
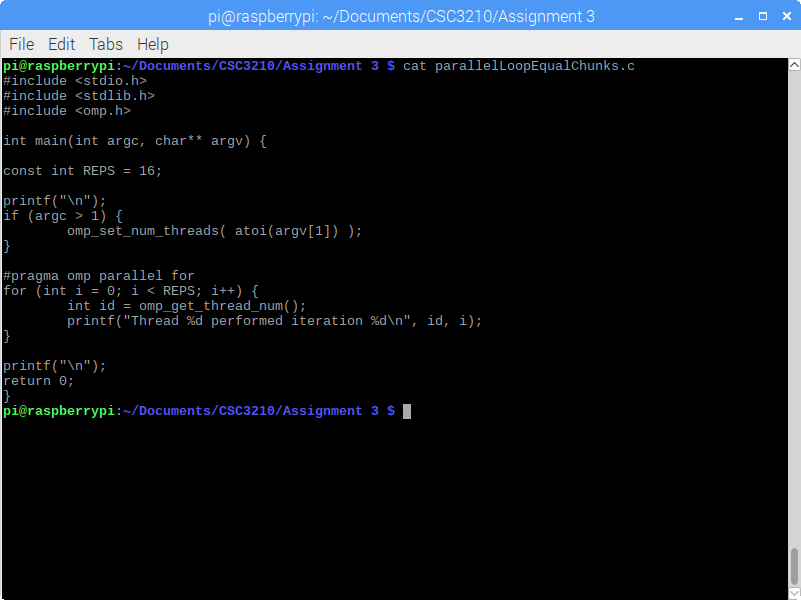
**Parallel Programming Task 2**

**1.1**

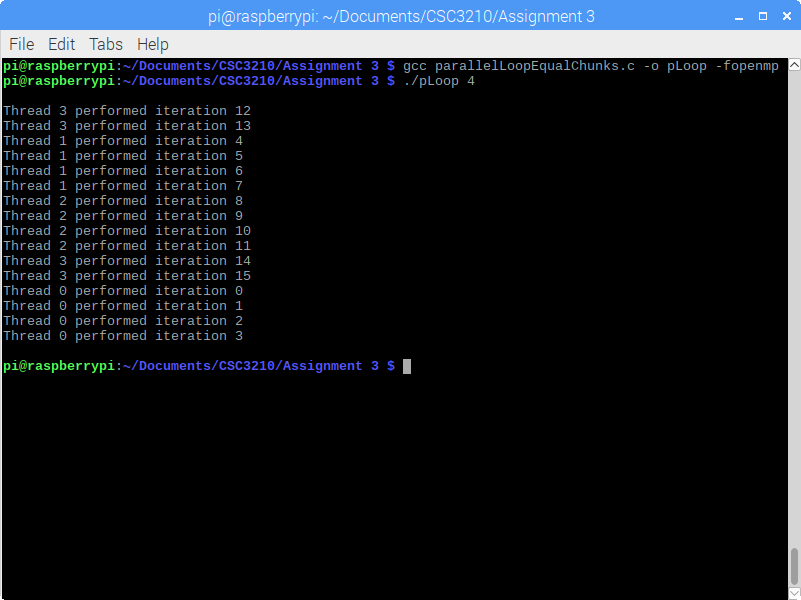
****

Terminal open in Raspberry Pi.

**2.**



The above screenshot shows the parallelLoopEqualChunks.c program in the terminal. (Note that though all programs in this lab report are shown using cat, they were written and edited using nano. Cat is only used to display them for the sake of space and readability.)

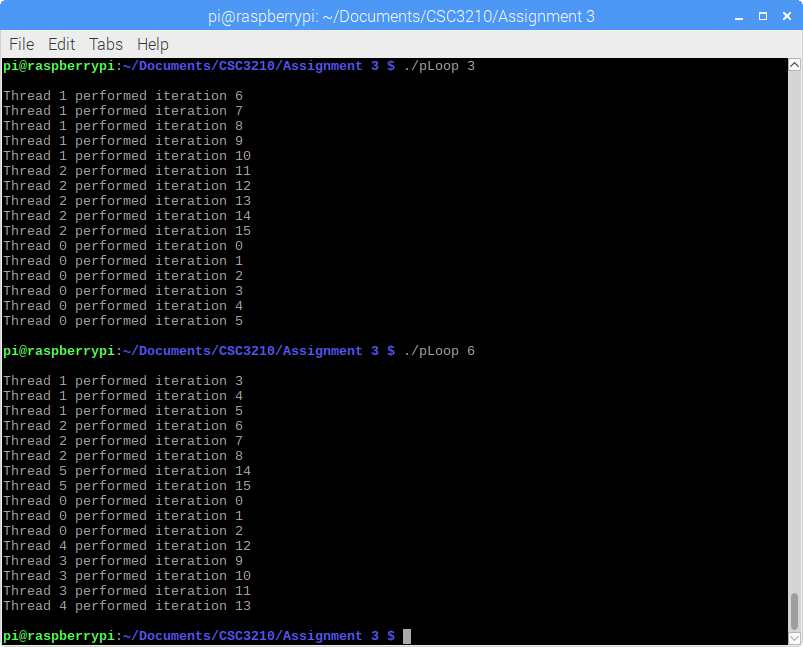


This screenshot shows the output given an input argument 4 of the parallelLoopEqualChunks.c program (Using 4 as the Raspberry Pi has 4 cores – running it without any argument defaults to 4 cores anyways.)

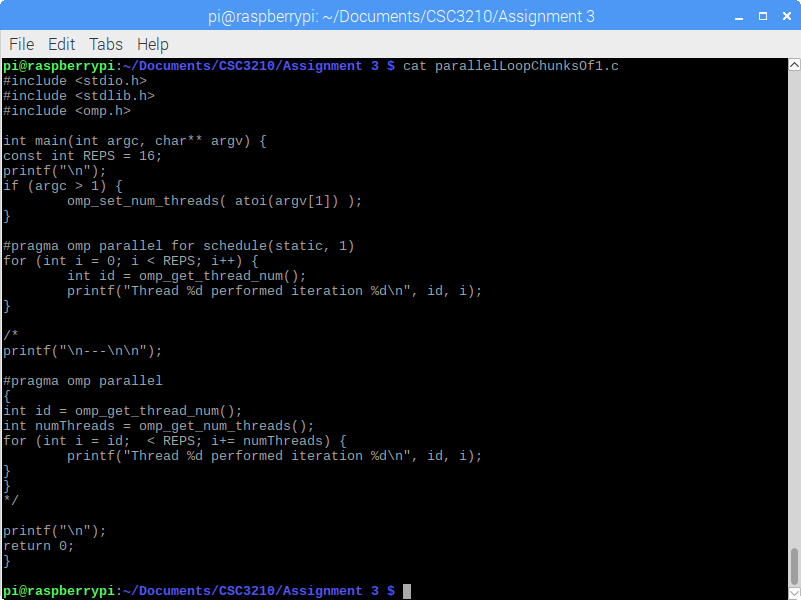
The parallelLoopEqualChunks.c program runs a total of 16 iterations of a loop across a given number of threads. As you can see, (though the threads themselves do not necessarily execute in any particular order), each of the four threads performs four *consecutive* iterations of the loop, with thread 0 performing the first four, thread 1 performing the next four, and so on. This is the default schedule data decomposition pattern for #pragma omp parallel for.

For a number of iterations N and a number of threads T, the number of iterations each thread will perform is N / T. This makes sense in this particular case, as 16 is evenly divisible by 4. But what if N is not evenly divisible by T? By the Quotient-Remainder theorem, any N can be represented as *d*\*T + *r*, where *d* = N/T (integer) and the remainder term *r* = N%T (which is at most T - 1). Therefore, whenever the number of iterations to be performed N is not divisible by the number of threads T, then each thread performs integer (N/T) – the *d* term – iterations, with the first (N%T) threads performing one additional iteration – the *r* term.

This can be seen in the screenshot below, tested using 3 and 6 threads, respectively.

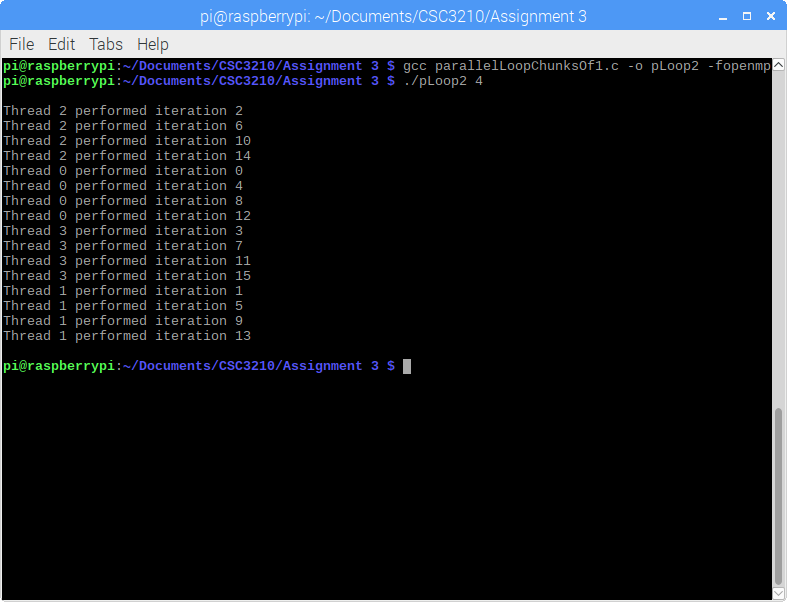


**3.**

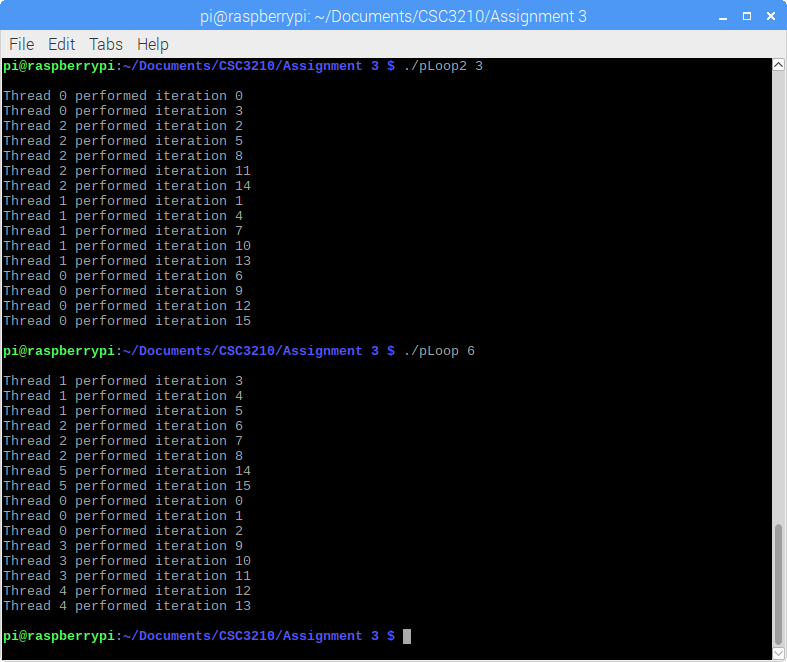
****

The above screenshot shows the parallelLoopChunksOf1.c program in the Raspbian terminal (once again, it is shown using cat here, but was written using nano.)

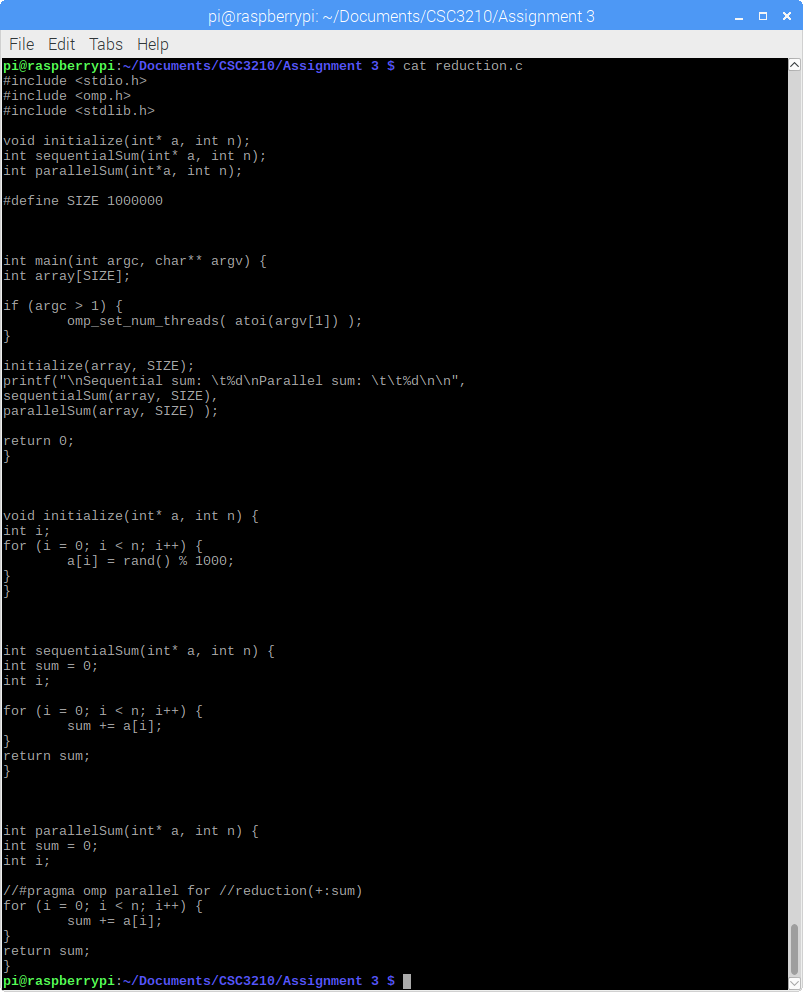
The fundamental difference between this program and the first is the schedule used for data decomposition. Take special note of the “#pragma omp parallel for” line. In this program, there is an additional clause “schedule(static , 1)”. This tells the compiler that instead of using the default data decomposition schedule, this uses a special static decomposition model. What this means is that instead of each thread having a prescribed set of consecutive iterations to run, each thread is given 1 iteration (this is the second parameter, can be changed to anything you like), and then the next thread is given the next iteration. In essence, this means that instead of performing the first (N/T) iterations, each thread performs every (N/T)th iteration. This can be seen in the output of the parallelLoopChunksOf1.c program shown below. Notice that thread 0 performs iterations 0, 4, 8, and 12, and thread 1 performs iterations 1, 5, 9, and 13, and so on.

****

Our earlier discussion of the quotient-remainder theorem and how it applies to the division of tasks applies naturally in the same way to this program. In general, each thread performs N/T iterations, with the overflow being given to the lowest threads. Once again, this can be seen in the screenshot below, tested with 3 and 6 threads.

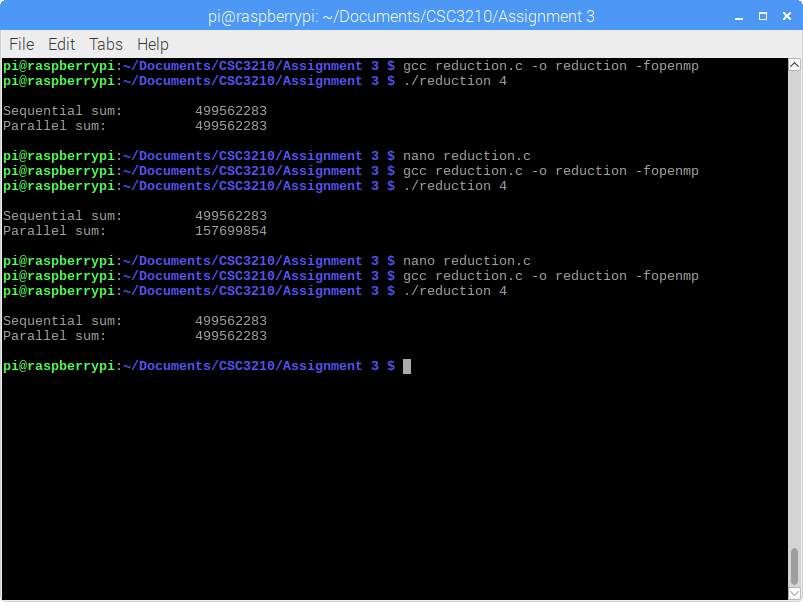
****

**4.**

****

The above screenshot shows the reduction.c program in the terminal.

This program initializes a large array with random values, and contains two methods: one to sum the elements of the array sequentially, and the other to sum the elements of the array using parallelization. It outputs the sum using both methods.

****

This screenshot shows the output of the reduction.c program after three edits. In each, the sum of the array is printed using the sequential method (which is useful to verify the results of the parallel method) and then the parallel method. In the first run, the “#pragma omp parallel” line of the parallel summing method is commented out. In this instance you can see that this method then functions essentially identically to the sequential method. In the second, the “#pragma omp parallel” line is uncommented, but only partially, leaving out the “reduction(+:sum)” clause. Here, you can see that both methods run, but the value of the parallel summing method is not correct. Notice, a hint here as to where the program went wrong is that the sum is roughly one quarter of the actual sum. Not unlike the issue of shared memory from the spmd2 program of the first assignment, each thread has its own sum variable. So, when the program reaches the “return sum;” statement, only one thread returns its own copy of the sum variable. This is why the “reduction(+:sum)” clause is necessary. This tells the compiler to add all of the individual sum variables.