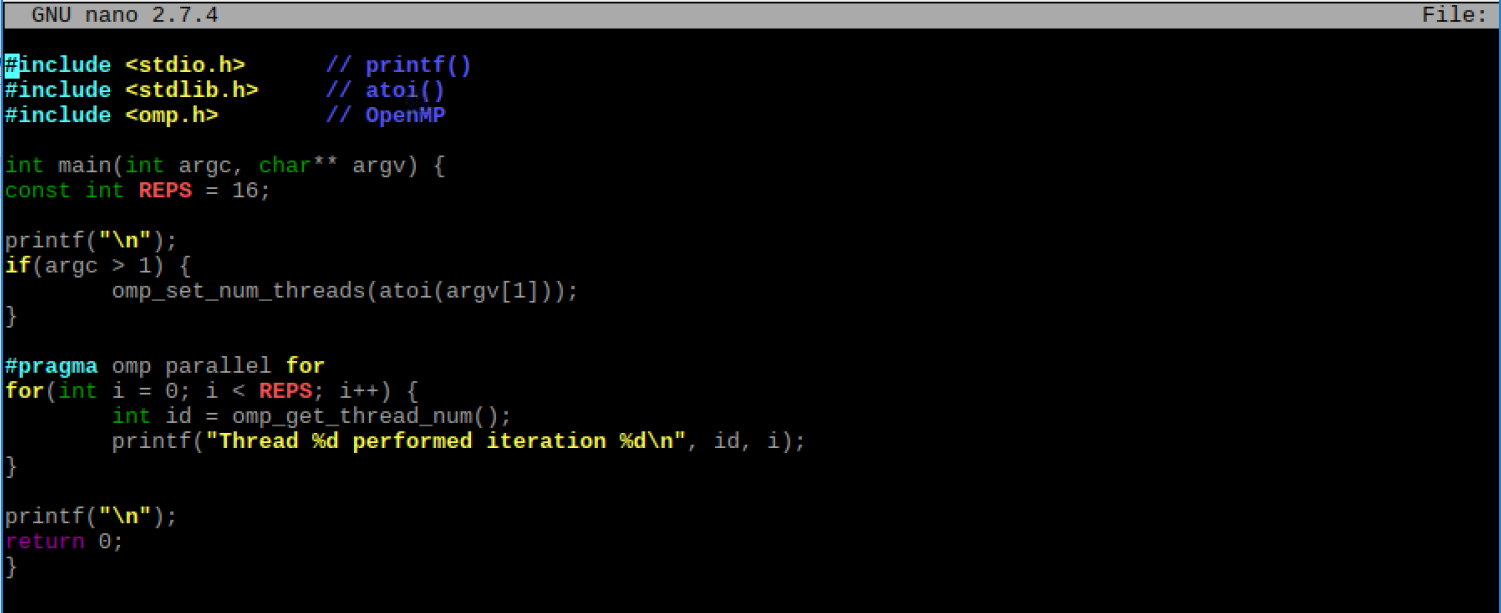
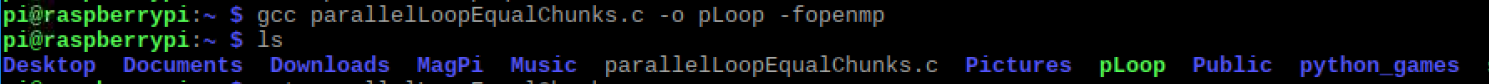
**Parallel Programming Basics –**

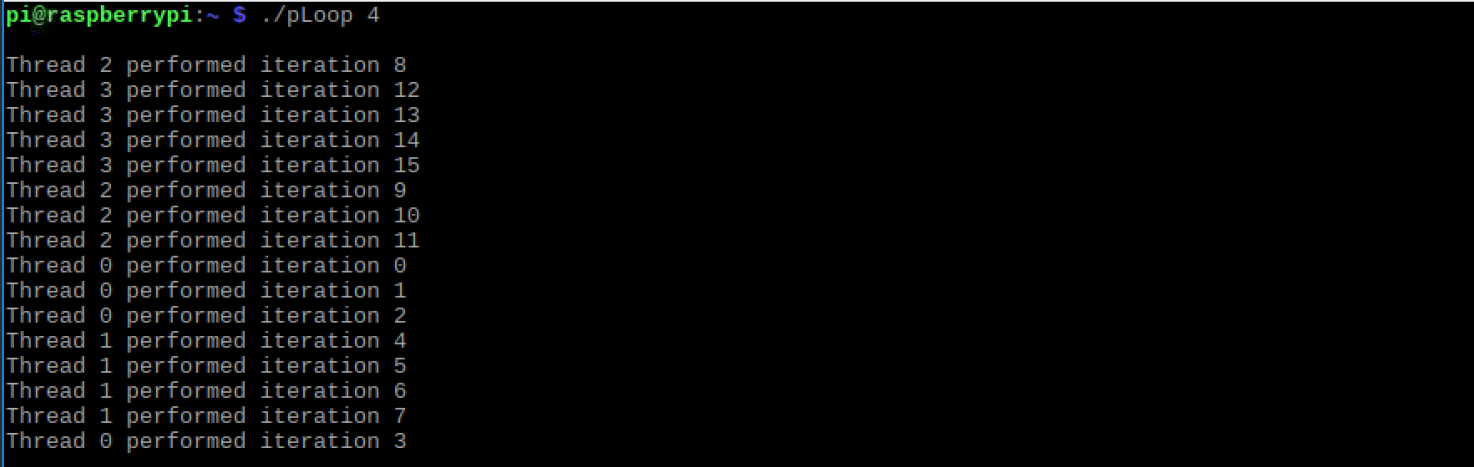
**Program 1: parallelLoopEqualChunks.c**

$ nano parallelLoopEqualChunks.c

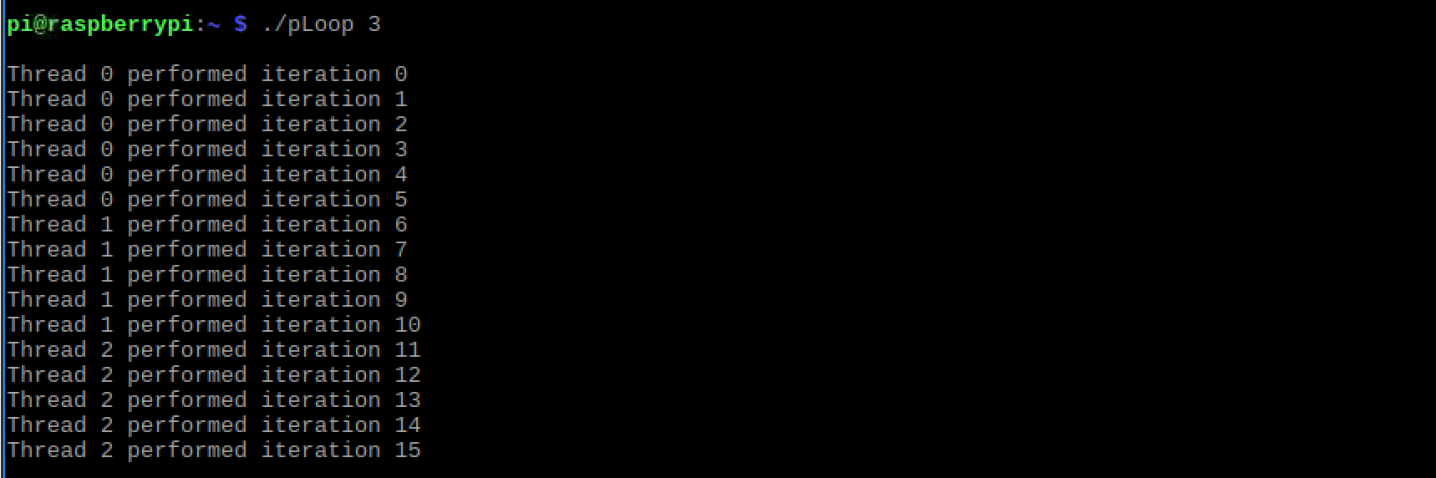
Here we typed out the C file in the terminal window using nano. To compile the program and rename it *pLoop*, we performed the following:

$ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp

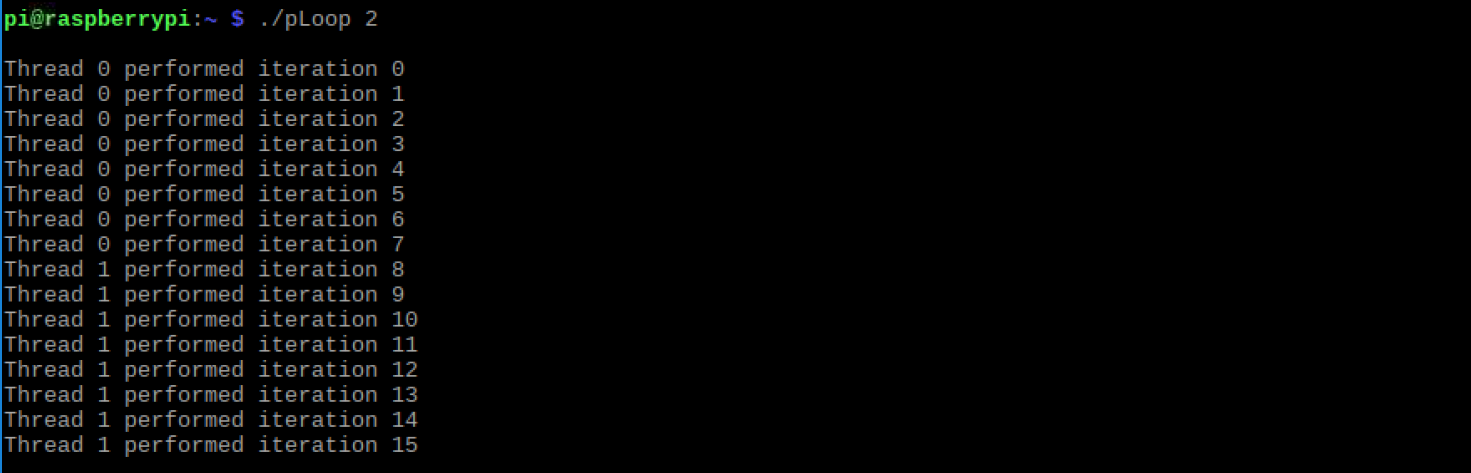
To make sure our file was created, we utilized the ‘ls’ utility. Then, we ran the program with 4 threads by using 4 as the argument. Since the Raspberry Pi has 4 cores, it was natural to try 4 threads.

$ ./pLoop 4

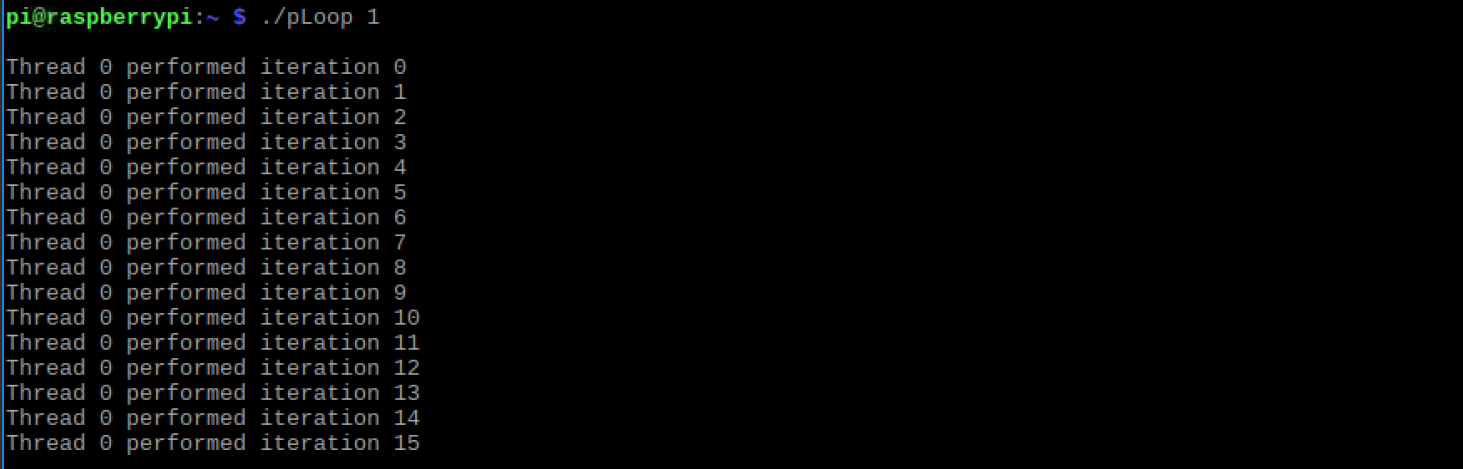
It is interesting to note that the order of the iterations as displayed on the terminal follows no specific pattern, with the exception of threads undertaking consecutive iterations. One would think that the iterations would move in order, or the order of when the thread executes, but this is not the case. The more important take away is that thread 0 manages the first 4 iterations (0-3), thread 1 manages the second 4 iterations (4-7), thread 2 manages the third 4 iterations (8-11), and thread 3 manages the last 4 iterations (12-15). Below we will run the same program using arguments 3, 2, 1 and no argument.

$ ./pLoop 3

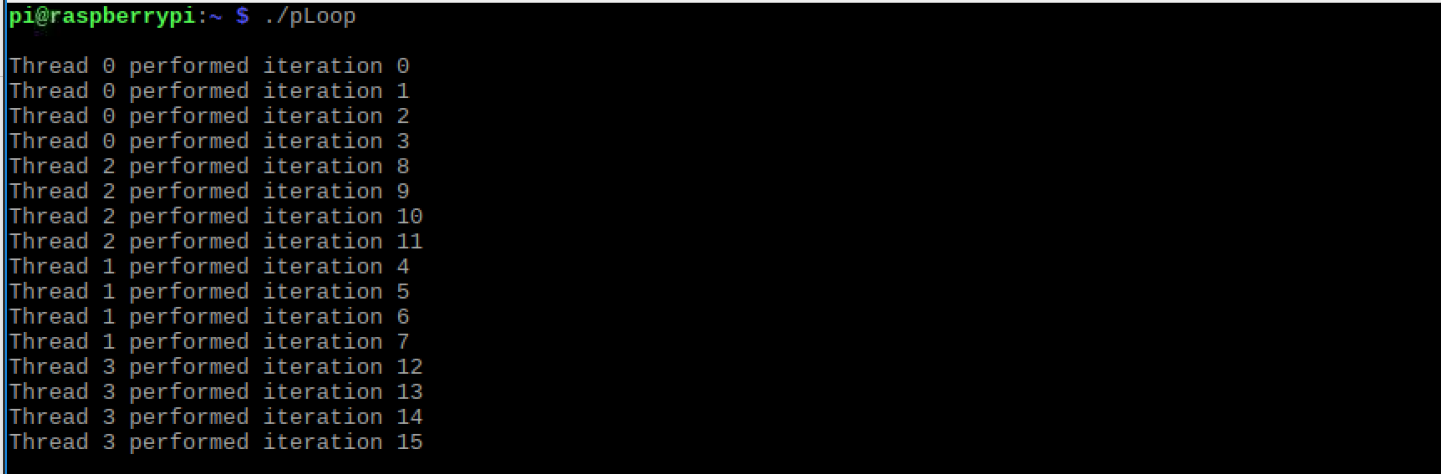
With 3 as the argument, thread 0 performed 6 iterations (0-5), thread 1 performed 5 iterations (6-10), and thread 2 performed 5 iterations (11-15). The interesting takeaway is that thread 0 had one additional iteration in comparison to the threads 1 and 2.

$ ./pLoop2

With 2 as the argument, thread 0 performed 8 iterations (0-7)and thread 1 performed 8 iterations (8-15). The total number of iterations was evenly divided among the 2 threads.

$ ./pLoop 1

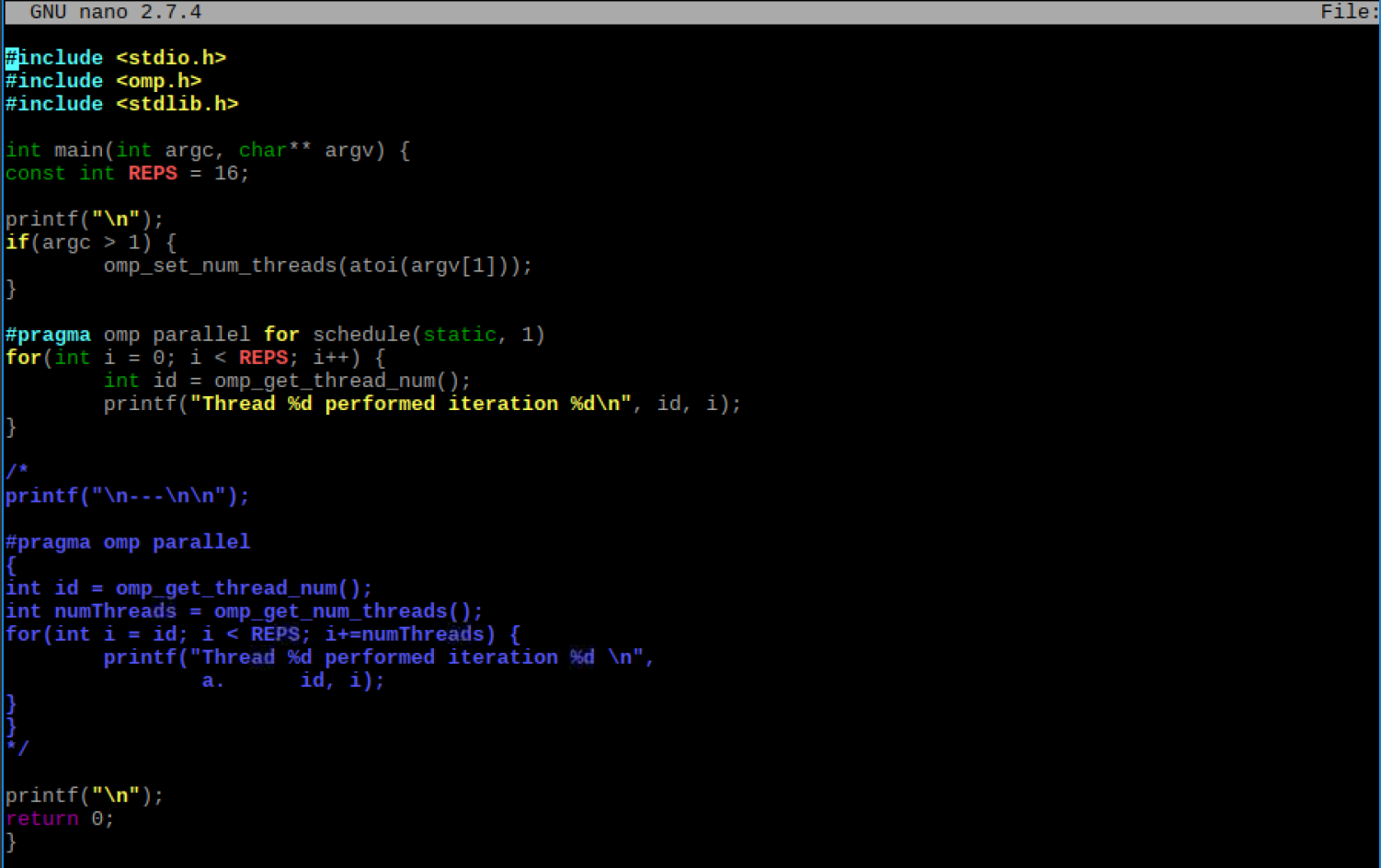
With 1 as the argument, thread 0 performed all iterations. This is not surprising.

$ ./pLoop

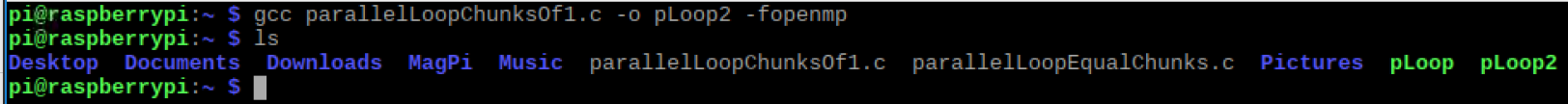
With no argument, it’s surprising to see that the threads were set according to the number of available cores (4 cores = 4 threads). Again, the output is like that of the output with 4 as the argument.

It appears that a modulo operator is used to calculate the remainder when the number of iterations is not evenly divisible by the number of threads. One iteration is added on to each lower thread until the remaining value (remainder = remaining iterations) is depleted. For example, in the *pLoop* program with 3 as the argument, thread 0 took on the additional iteration. We know we have 16 iterations, and we have 3 as the argument. 16 mod 3 = 1. With 1 as the remainder, this was added onto the lowest thread, thread 0, giving it 6 iterations; while the other threads undertook 5 iterations.

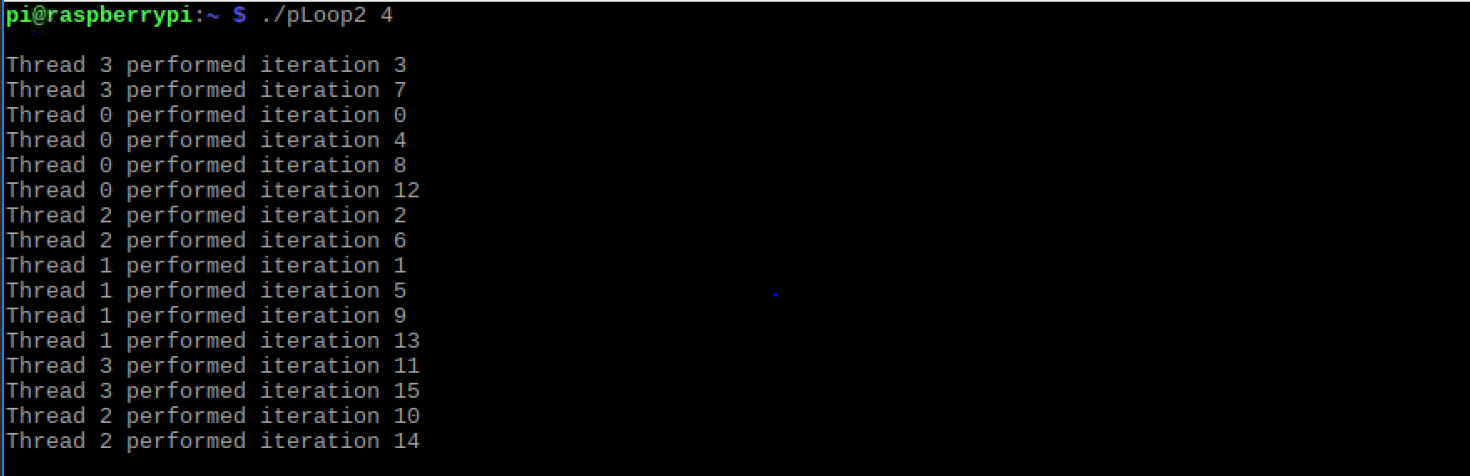
**Program 2: parallelLoopChucksof1.c**

$ nano parallelLoopChunksof1.c

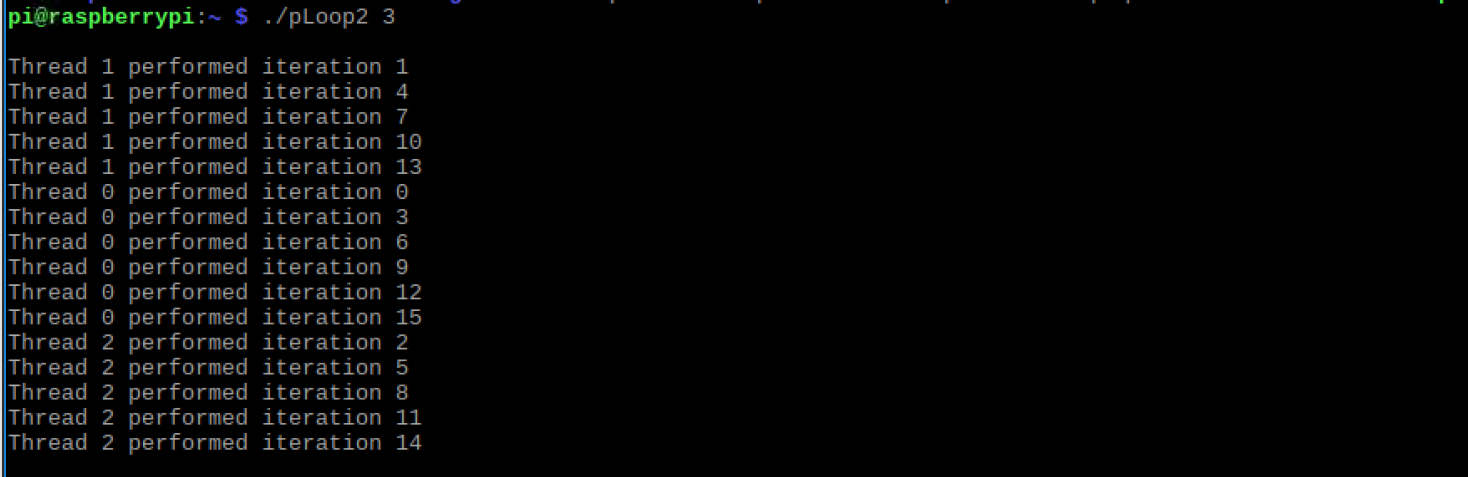
Here we typed out the C file in the terminal window using nano. To compile the program and rename it *pLoop2*, we performed the following:

$ gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp

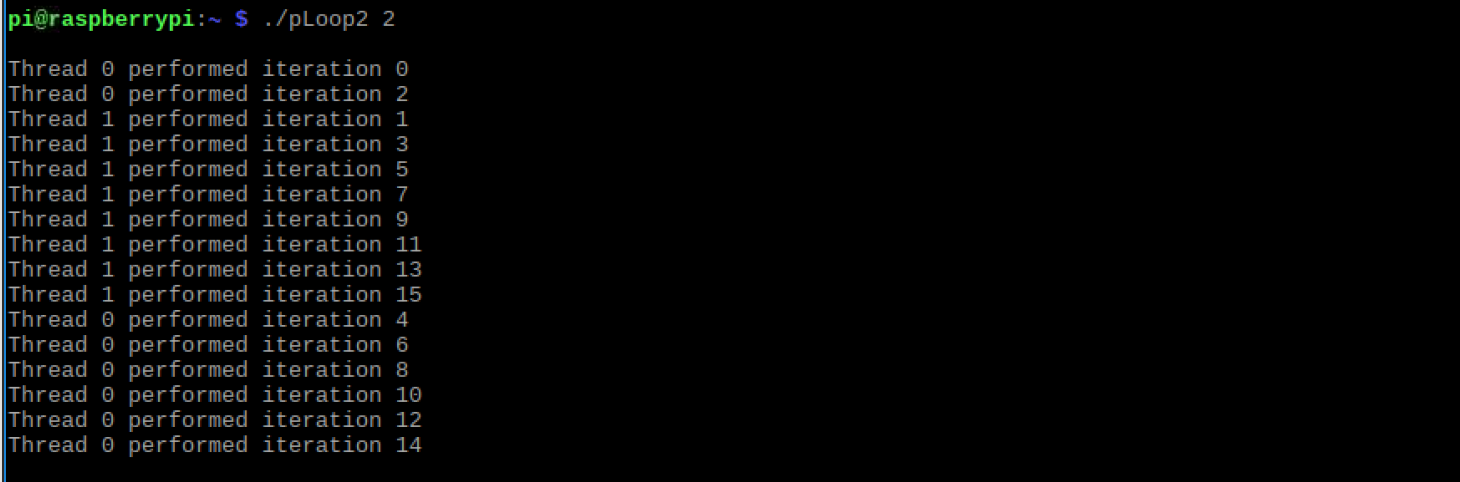
To make sure our file was created, we utilized the ‘ls’ utility. Then, we ran the program with 4 threads by using 4 as the argument. Since the Raspberry Pi has 4 cores, it was natural to try 4 threads.

$ ./pLoop2 4

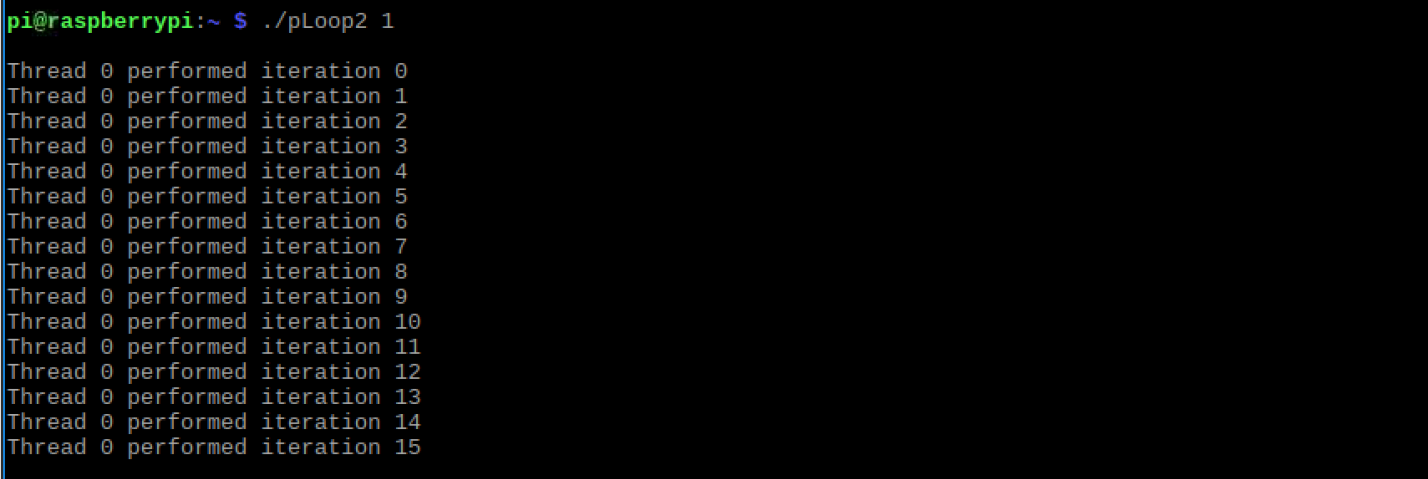
After running the program with an argument of 4, we can see that each thread computes an equal amount of work, just not consecutive iterations. In this case, each thread completes 1 (chunk size) iteration and moves on to the next thread/iteration combination (i.e. 0 thread performs iteration 0, 1 thread performs iteration 1, thread 2 performs iteration 2, and so on until the 4 threads complete the first 4 iterations; then repeats until all iterations are complete). This is surprising in the sense that declaring a static assignment can dictate the amount of work and order by which threads can compute work. Below we will replace 4 with other values for the number of threads.

./pLoop2 3

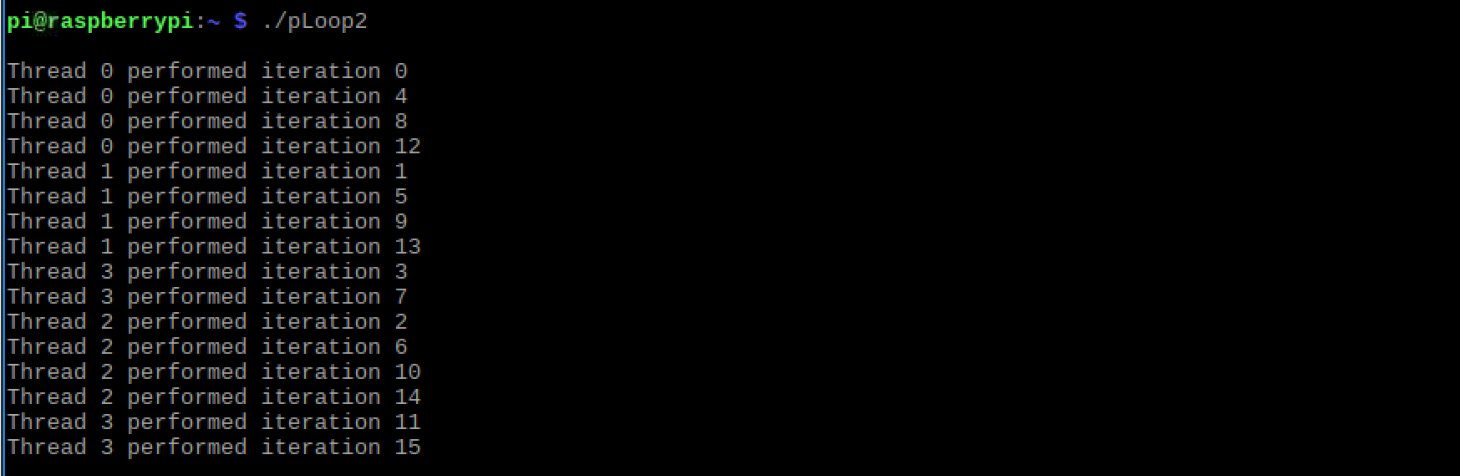
With 3 as the argument, we can see the number of threads is set to 3 threads (threads 0-2). With static being set and assigned a value of 1 (chunk size), we schedule each thread to do one iteration of the loop in a regular pattern. Also, we notice that the remainder principle still applies. Thread 0 has 6 iterations (0, 3, 9, 12, and 15), thread 1 has 5 iterations (1, 4, 7, 10, and 13), and thread 2 has 5 iterations (2, 5, 8, 11, and 14). Since 16 % 3 = 1, the remaining value 1 was added to thread 0—the lowest thread—giving it an extra iteration.

./pLoop2 2

With 2 as the argument, we see the same process with an equal number of iterations between the two threads 0 and 1.

./pLoop2 1

Thread 0 handles all the iterations. Although static and chunk size are set, there is only 1 thread which doesn’t meet the argument size requirement for the *if-statement*.

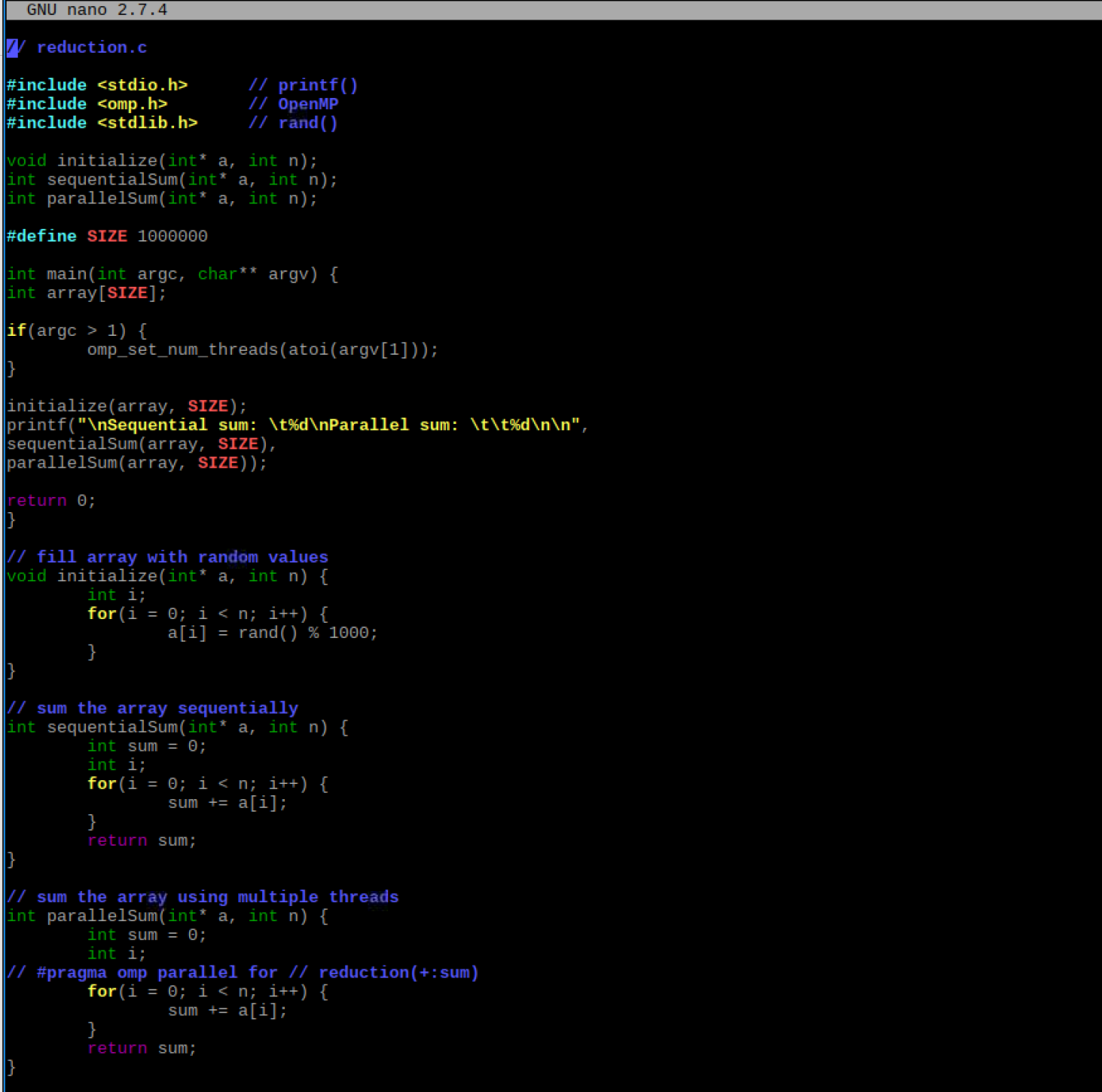
./pLoop2

The output of the *pLoop2* program without arguments is similar to the output when the argument is set to 4 (4 cores = 4 threads). This result is expected.

We can also assign the threads dynamically by changing *static* to *dynamic*. Under this assignment, when a thread completes it gets the next iteration or chuck still needed to be completed. For example:

#pragma omp parallel for schedule (**dynamic**, 1)

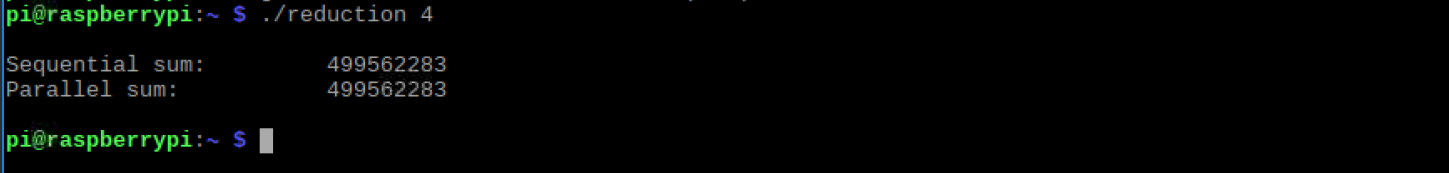
**Program 3: reduction.c**

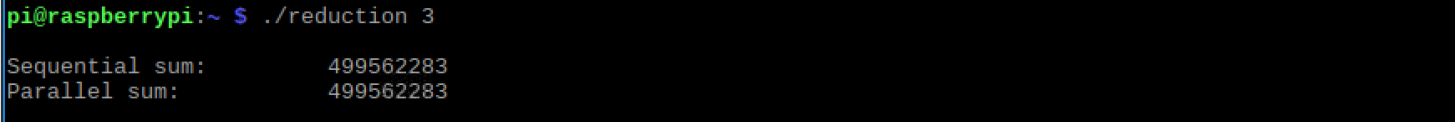
$ nano reduction.c

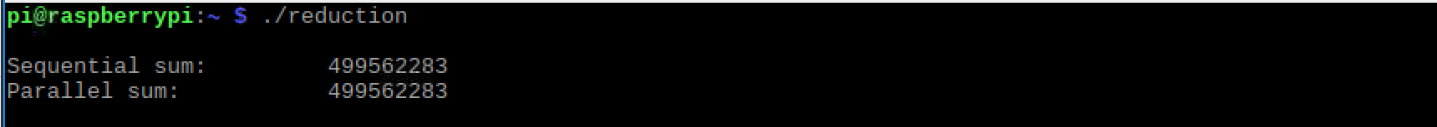
Here we typed out the C file in the terminal window using nano. To compile the program and rename it *reduction*, we performed the following:

$ gcc reduction.c -o reduction -fopenmp

After compiling we ran the program with arguments 4, 3, and no arguments. This is shown below.

./reduction 4

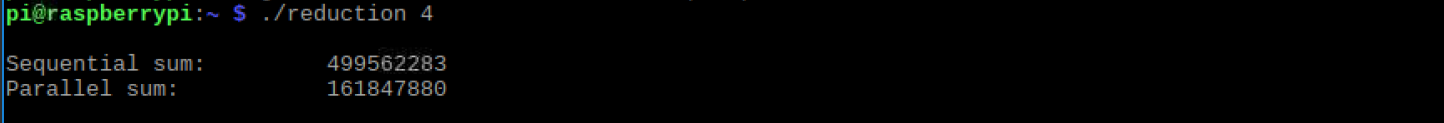
/.reduction 3

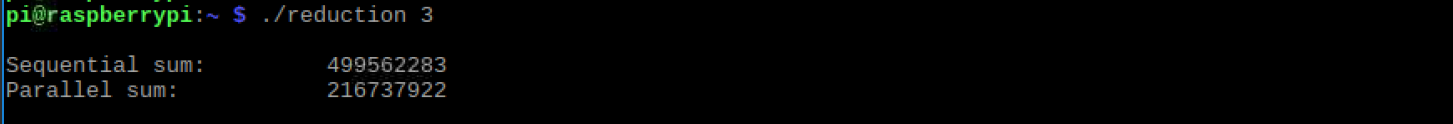
/.reduction

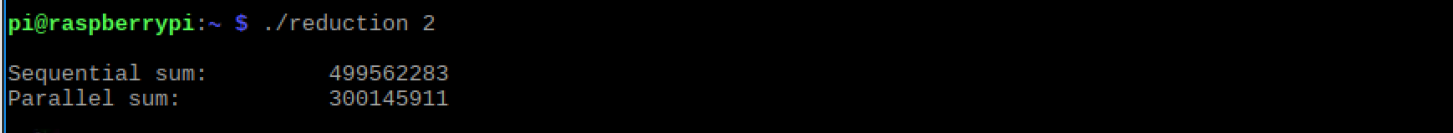
In this program, an array of randomly assigned integers is a set of data shared between both sequentialSum() and parallelSum() functions. The functions are identical while the #pragma line is commented out. Thus, we can expect the for loops to sum up all the values in the array. This results in the same output for each function. The correct output is produced by both sequentialSum() and parallelSum() in this example.

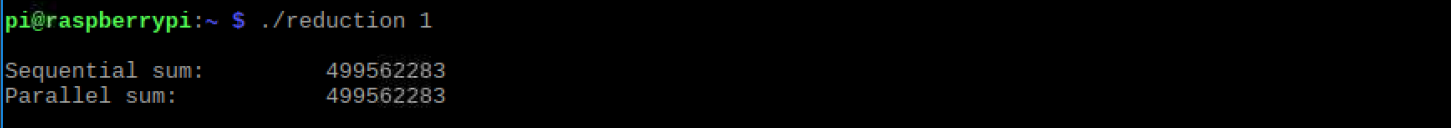
Now we will remove the // in front of #pragma in line 39, re-compile, and re-run the program. This will uncomment the first part of the #pragma line allowing the program to run in parallel. The output after running the program with arguments 4, 3, 2, 1, and no arguments are below:

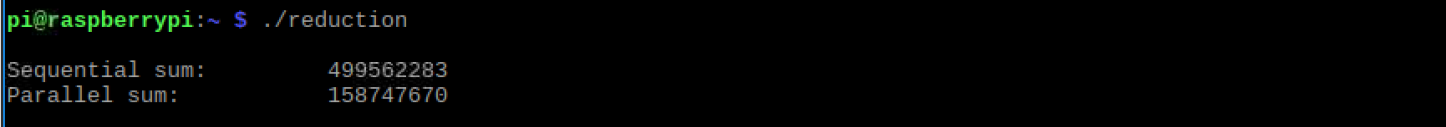
$ ./reduction *n* ;(where n = 4, 3, 2, 1, and “ “)







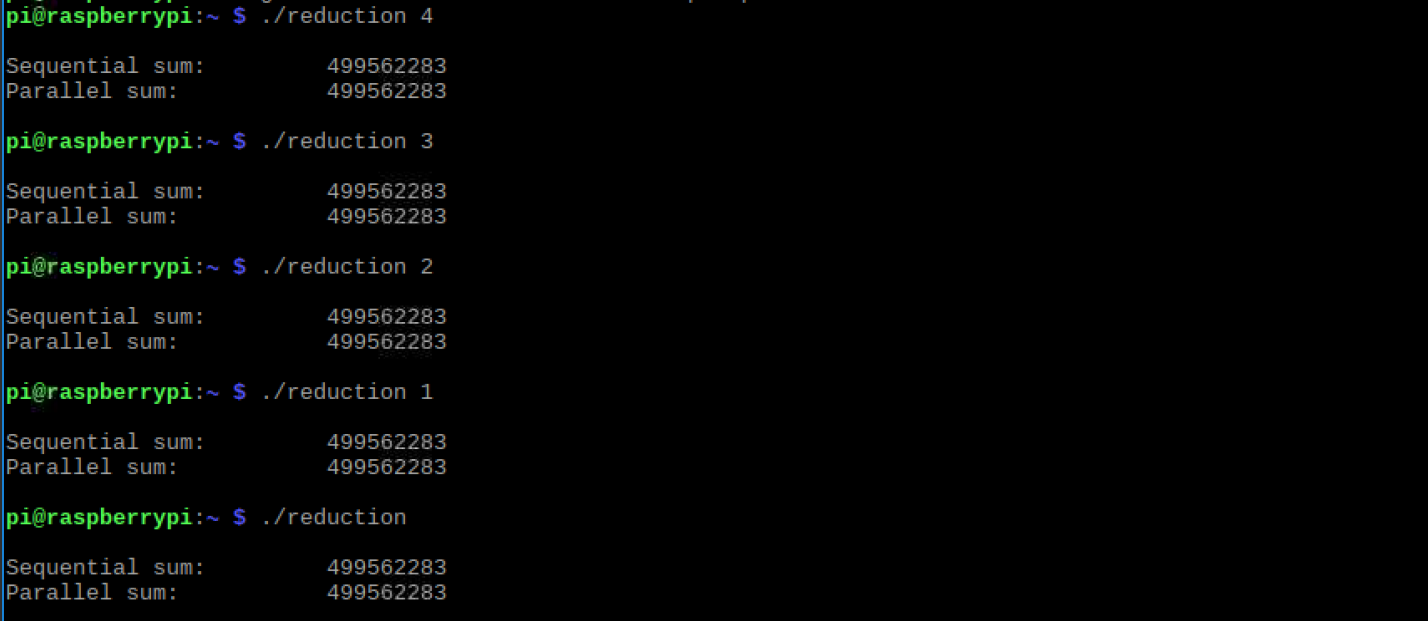




Here we immediately notice the sequentialSum() function’s results do not match the parallelSum() function’s results, apart from when the argument is set to 1. This makes sense because to set the number of threads, the argument must be a value greater than 1. Here parallelSum() does not produce the correct output. This is interesting as we are allowing parallel computation; however, without the reduction clause the threads cannot implicitly communicate to keep the overall sum updated as each work on a portion of the array.

Next we removed the second // in line 39, re-compiled and re-ran the program using arguments of 4, 3, 2, 1, and no argument. The output is shown below.

$ ./reduction *n* (where n = 4, 3, 2, 1, “ “)



The resulting sum of both functions now equals the correct output. This is what should happen as the reduction clause is now available to be read and executed. The reduction clause, reduction(+:sum), allows all the values to be summed together by using the OpenMP parallel for pragma. The plus sign in the pragma reduction clause signals the variable sum is being computed by added values together in the loop. The threads are now communicating to keep the overall sum updated as each of them works on the portion of the array.

We believe the parallel for pragma did not produce the correct result without the reduction clause because the threads have no way of communicating to track the sums of each iteration. This communication is paramount to return a sum of all computations performed by each thread. The accumulator sum needs to be private as each thread completes its work. Then, when each thread is finished, the final sum of their individual sums is computed. The variable sum is dependent on what the other threads are doing.

A full snapshot of program 3 is below.

