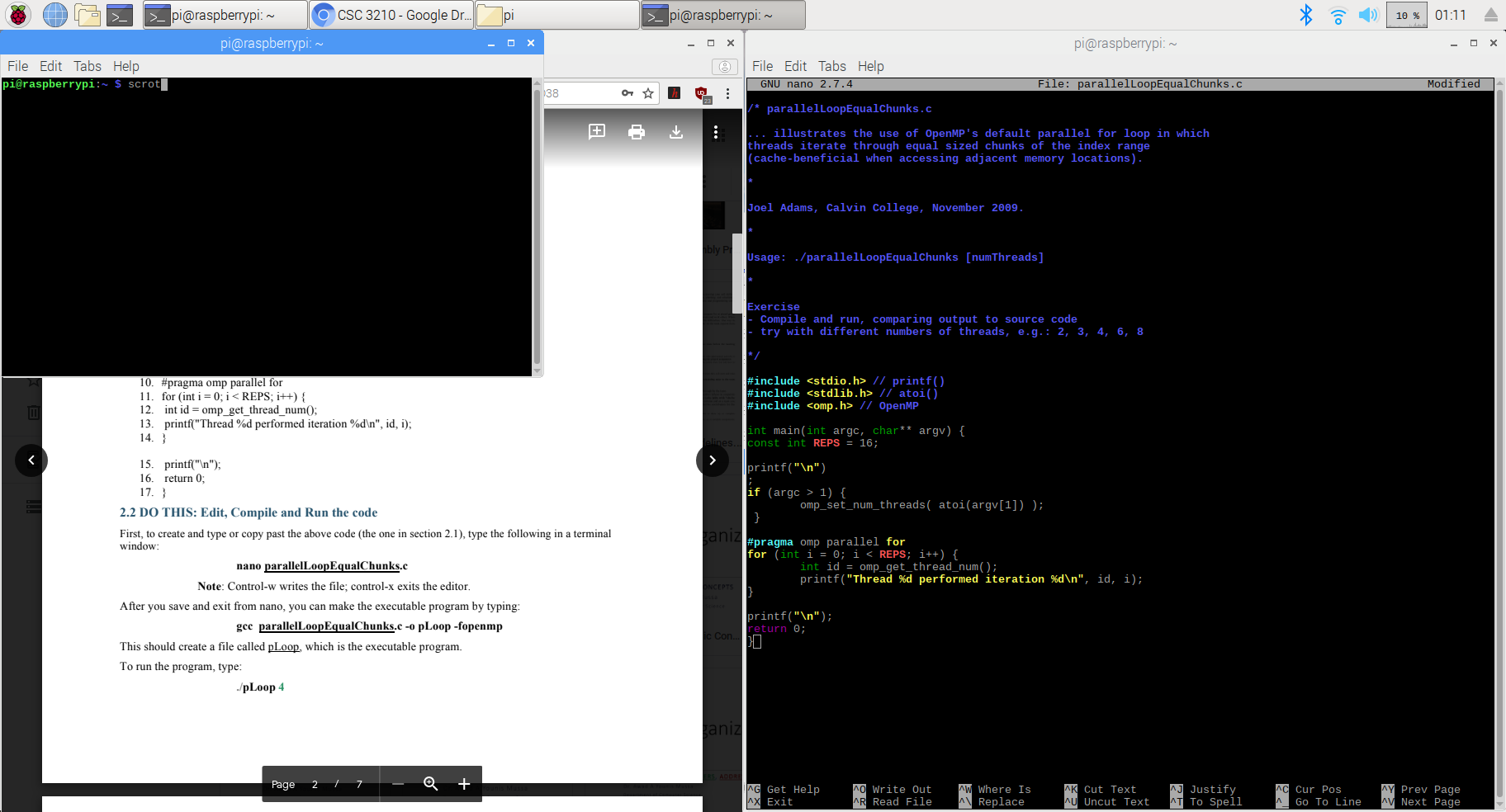
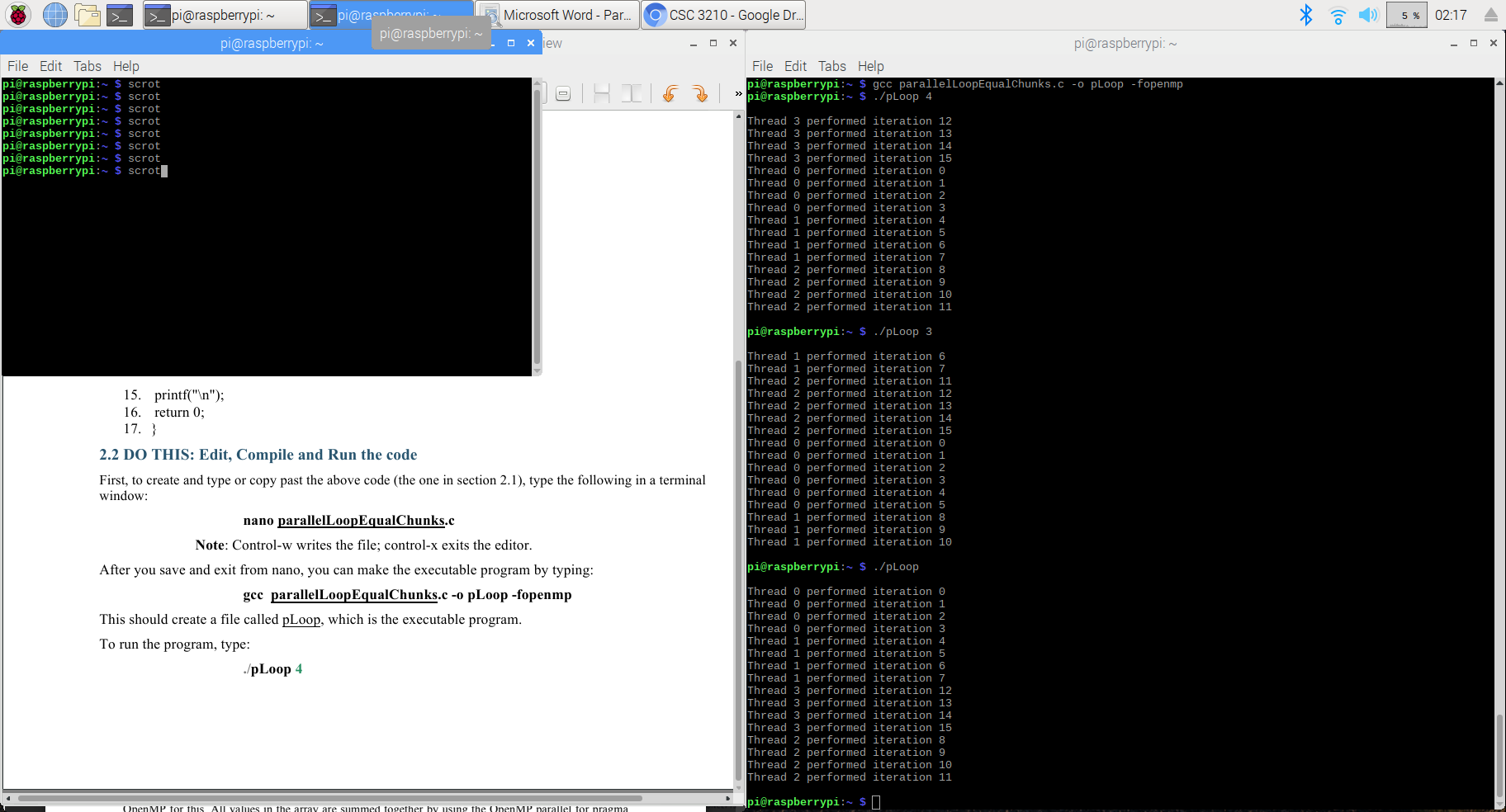
**A3 Parallel Programming – Team Hustle**





2.1 Running Loops in Parallel

* nano parallelLoopEqualChunks.c
* Code copied and pasted in the file
* Divides the work into consecutive iterations of the loop
* gcc parallelLoopEqualChunks.c -o pLoop -fopenmp

1. ./pLoop 4 (4 threads)

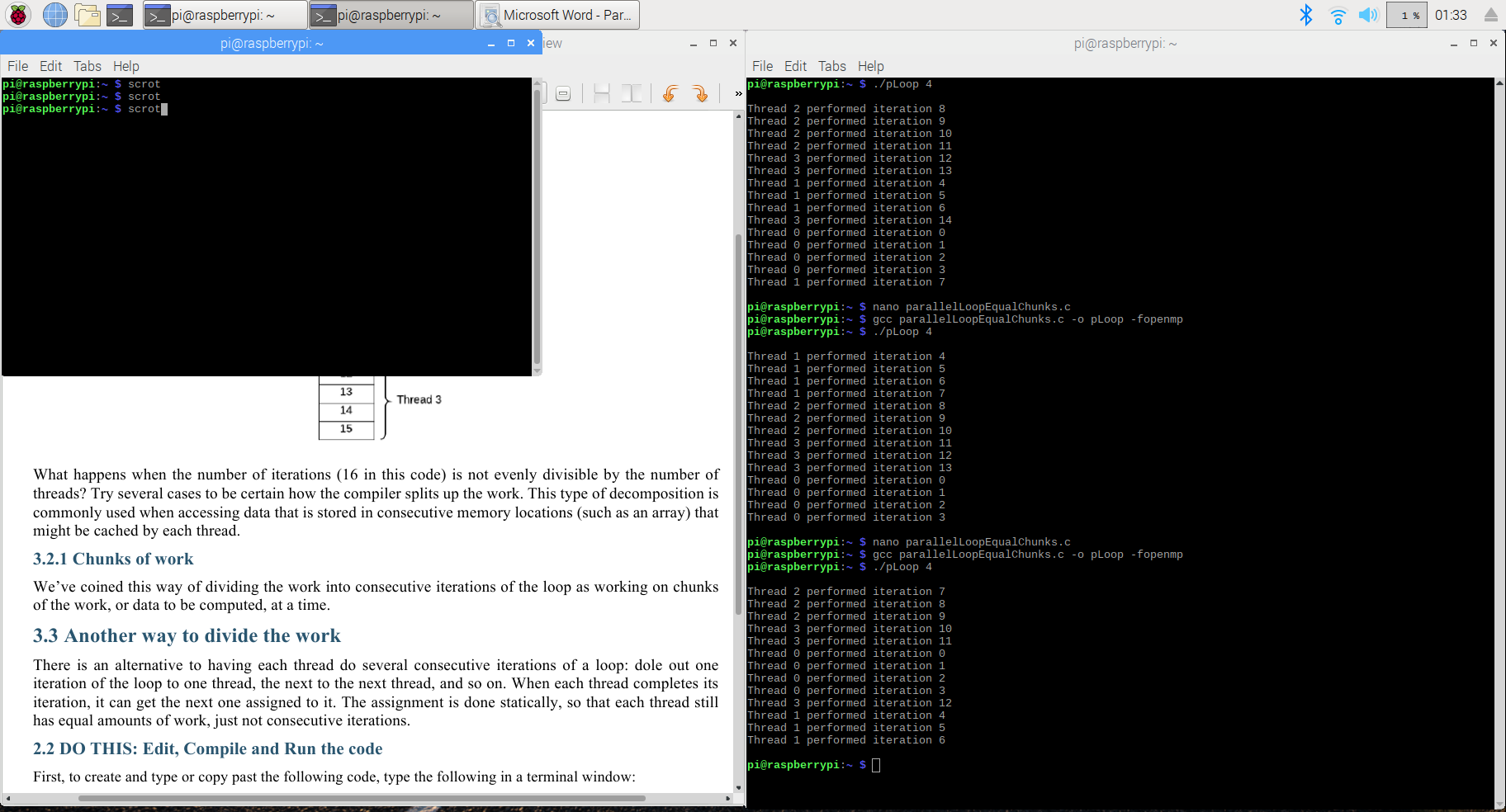
* Thread 0 takes iterations 0-3 (4)
* Thread 1 takes iterations 4-7 (4)
* Thread 2 takes iterations 8-11 (4)
* Thread 3 takes iterations 12-15 (4)

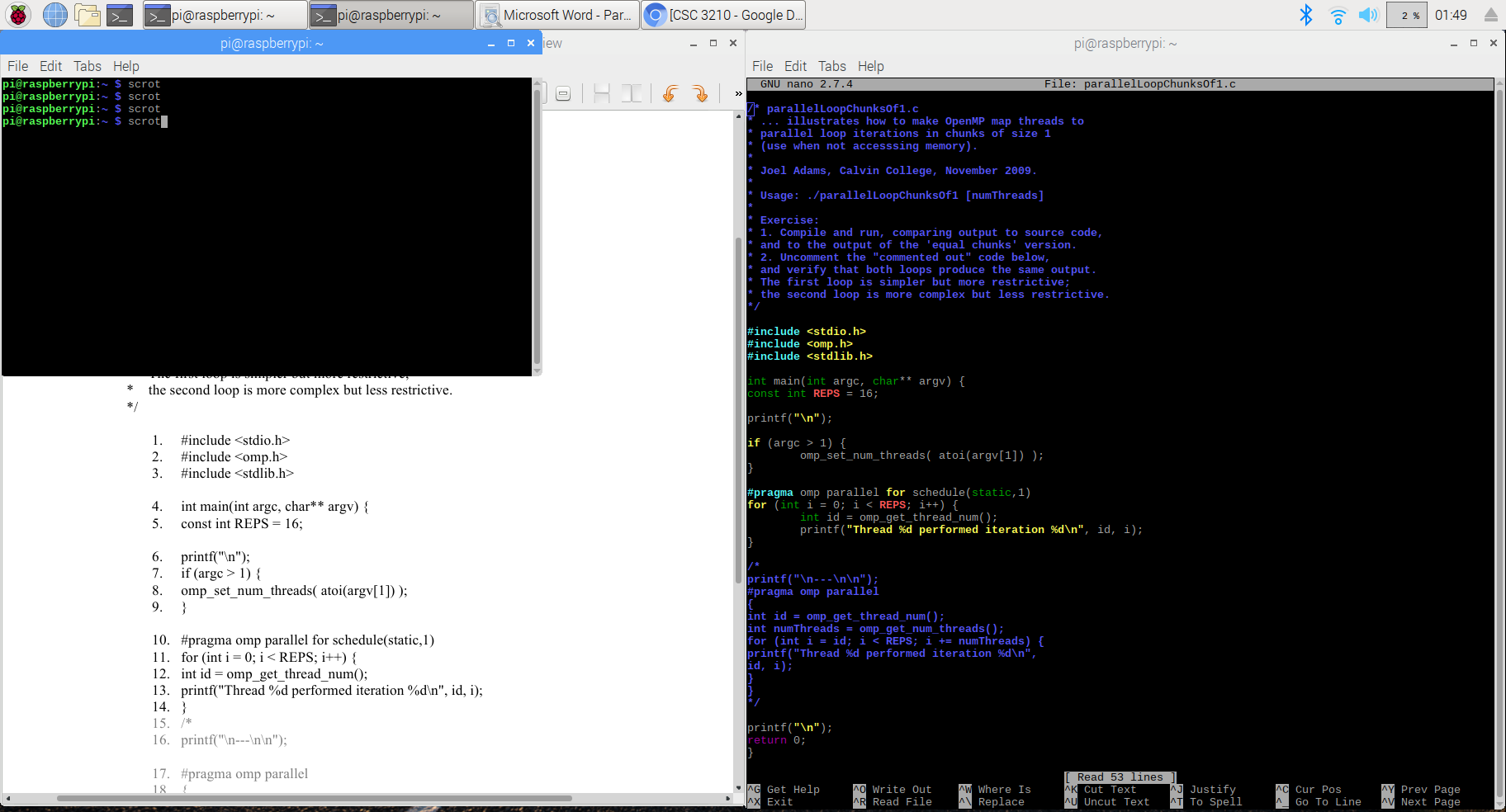
1. ./pLoop 3 (3 threads)

* Thread 0 takes iterations 0-5 (6)
* Thread 1 takes iterations 6-10 (5)
* Thread 2 takes iterations 11-15 (5)

1. ./pLoop (the number of threads not specified)

* 4 threads by default





* Changing the number of iterations

1. const int REPS = 15;

* Thread 0 takes iterations 0-3 (4)
* Thread 1 takes iterations 4-7 (4)
* Thread 2 takes iterations 8-11 (4)
* Thread 3 takes iterations 12-14 (3)

1. const int REPS = 14;

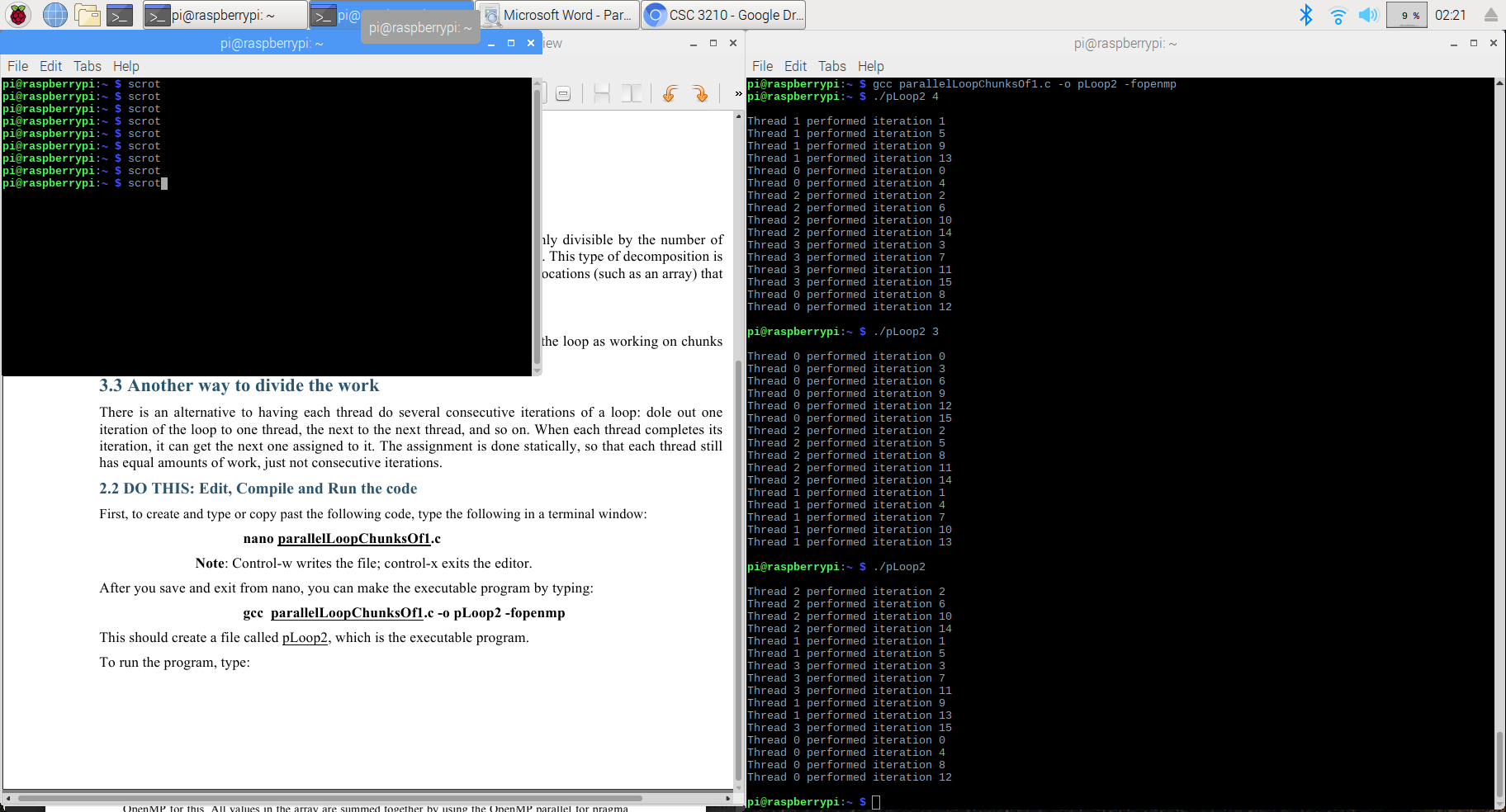
* Thread 0 takes iterations 0-3 (4)
* Thread 1 takes iterations 4-7 (4)
* Thread 2 takes iterations 8-10 (3)
* Thread 3 takes iterations 11-13 (3)

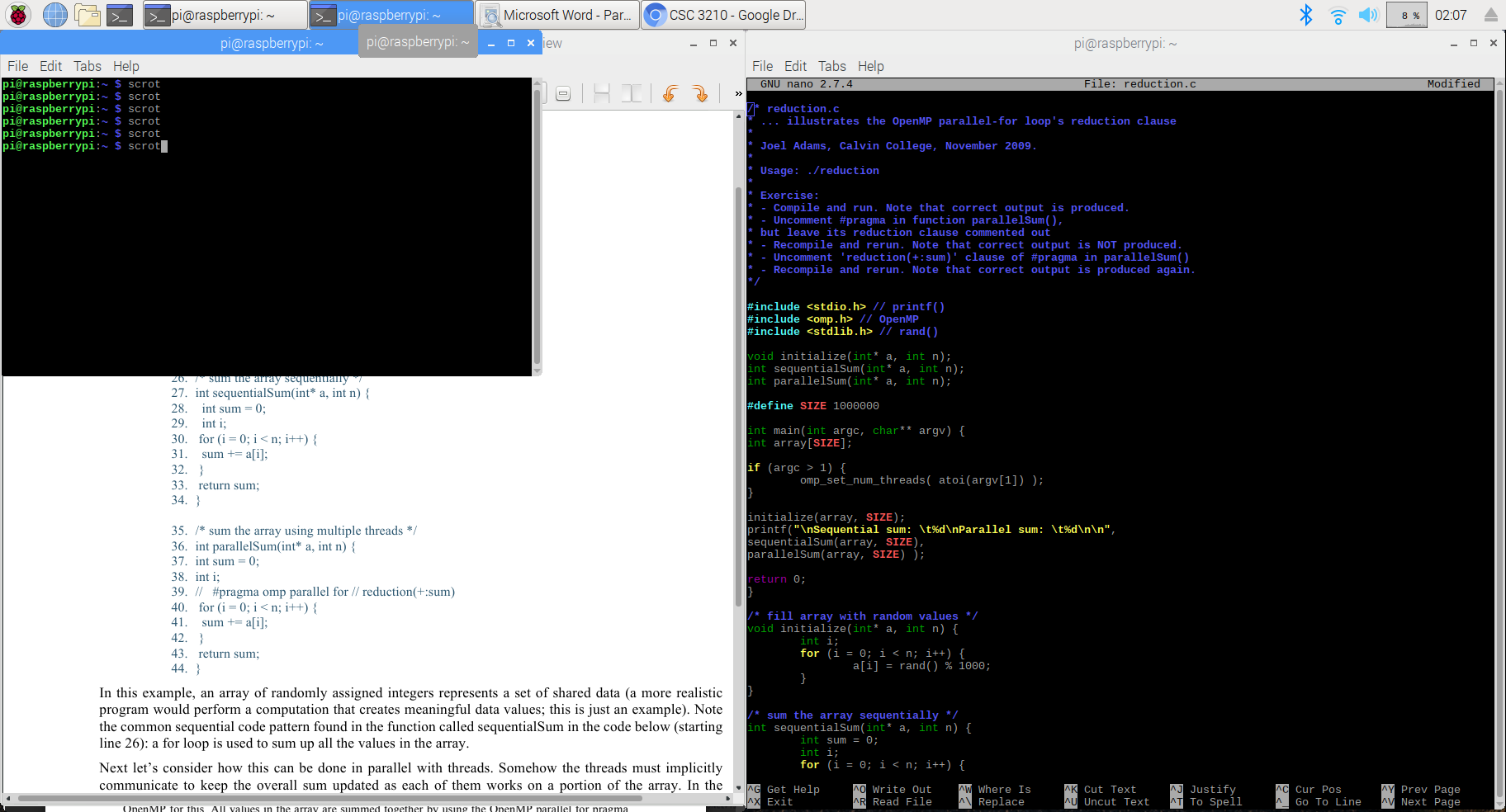
1. const int REPS = 13;

* Thread 0 takes iterations 0-3 (4)
* Thread 1 takes iterations 4-6 (3)
* Thread 2 takes iterations 7-9 (3)
* Thread 3 takes iterations 10-12 (3)

3.3 Another Way to Divide the Work

* nano parallelLoopChunksOf1.c
* Code copied and pasted in the file
* Gives one iteration of the loop to one thread, the next to the next thread, and so on.





* gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp

1. ./pLoop2 4 (4 threads)

* Iterations 0-3 taken by threads 0-3, respectively
* Iterations 4-7 by threads 0-3
* Iterations 8-11 by threads 0-3
* Iterations 12-15 by threads 0-3

1. ./pLoop2 3 (3 threads)

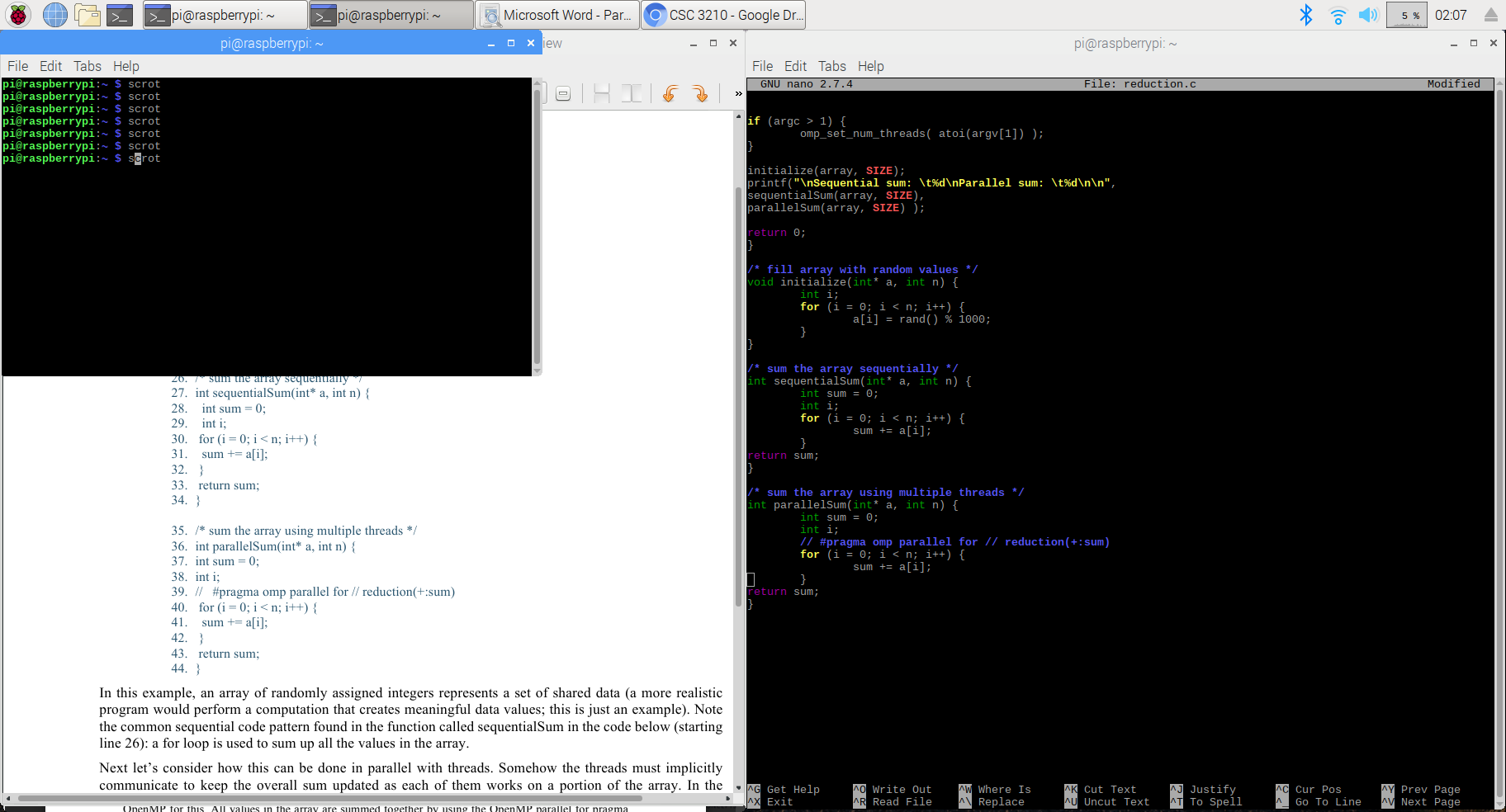
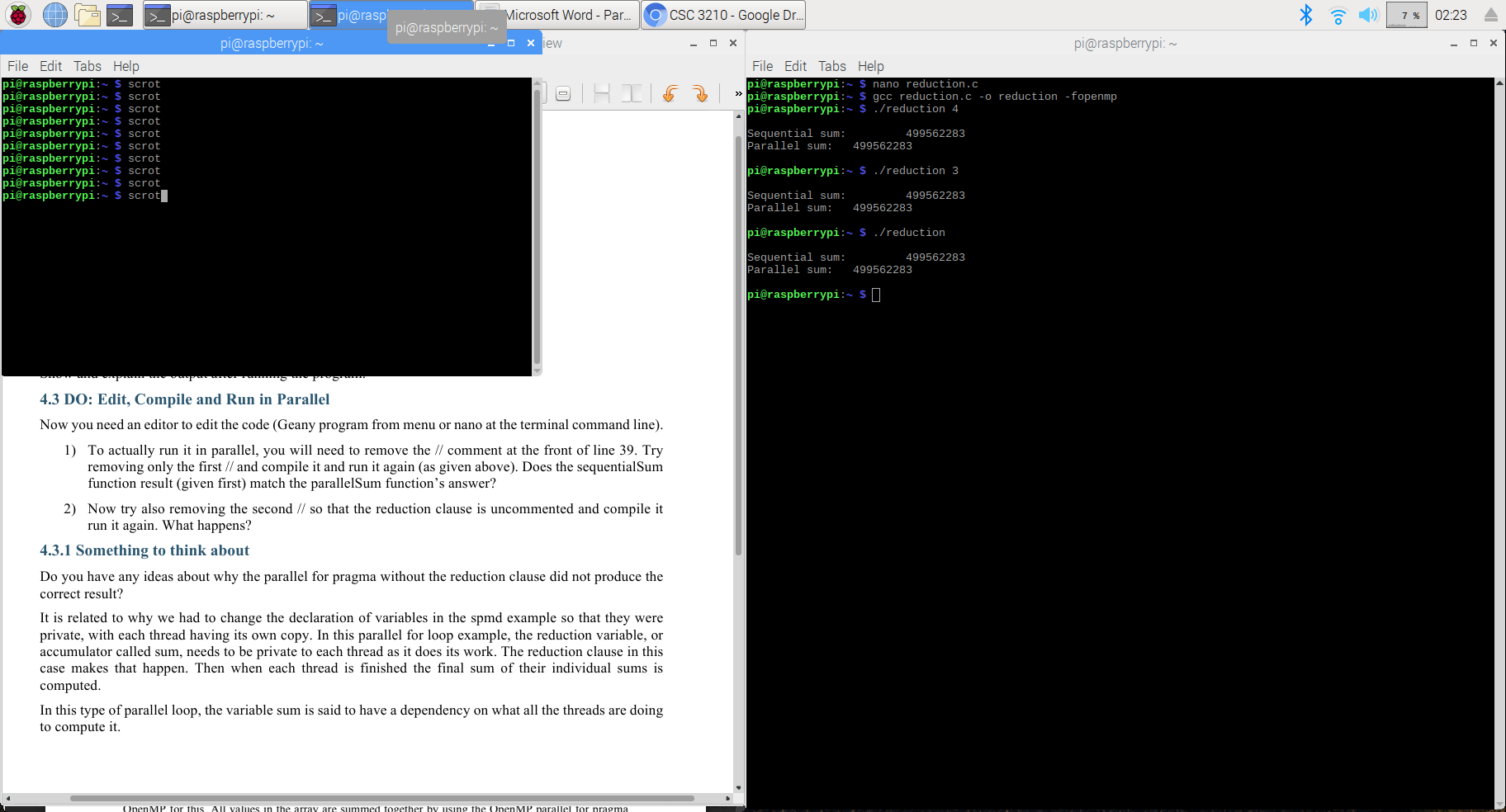
* Iterations 0-2 taken by threads 0-2, respectively
* Iterations 3-5 by threads 0-2
* Iterations 6-8 by threads 0-2
* Iterations 9-11 by threads 0-2
* Iterations 12-14 by threads 0-2
* Iteration 15 by thread 0

1. ./pLoop2 (the number of threads not specified)

* 4 threads by default

4.1 When Loops Have Dependencies

* nano reduction.c
* Code copied and pasted in the file
* Takes an array of randomly assigned integers and calculate the sum of all values in the array in two ways: one in sequential computation and another in parallel with threads.

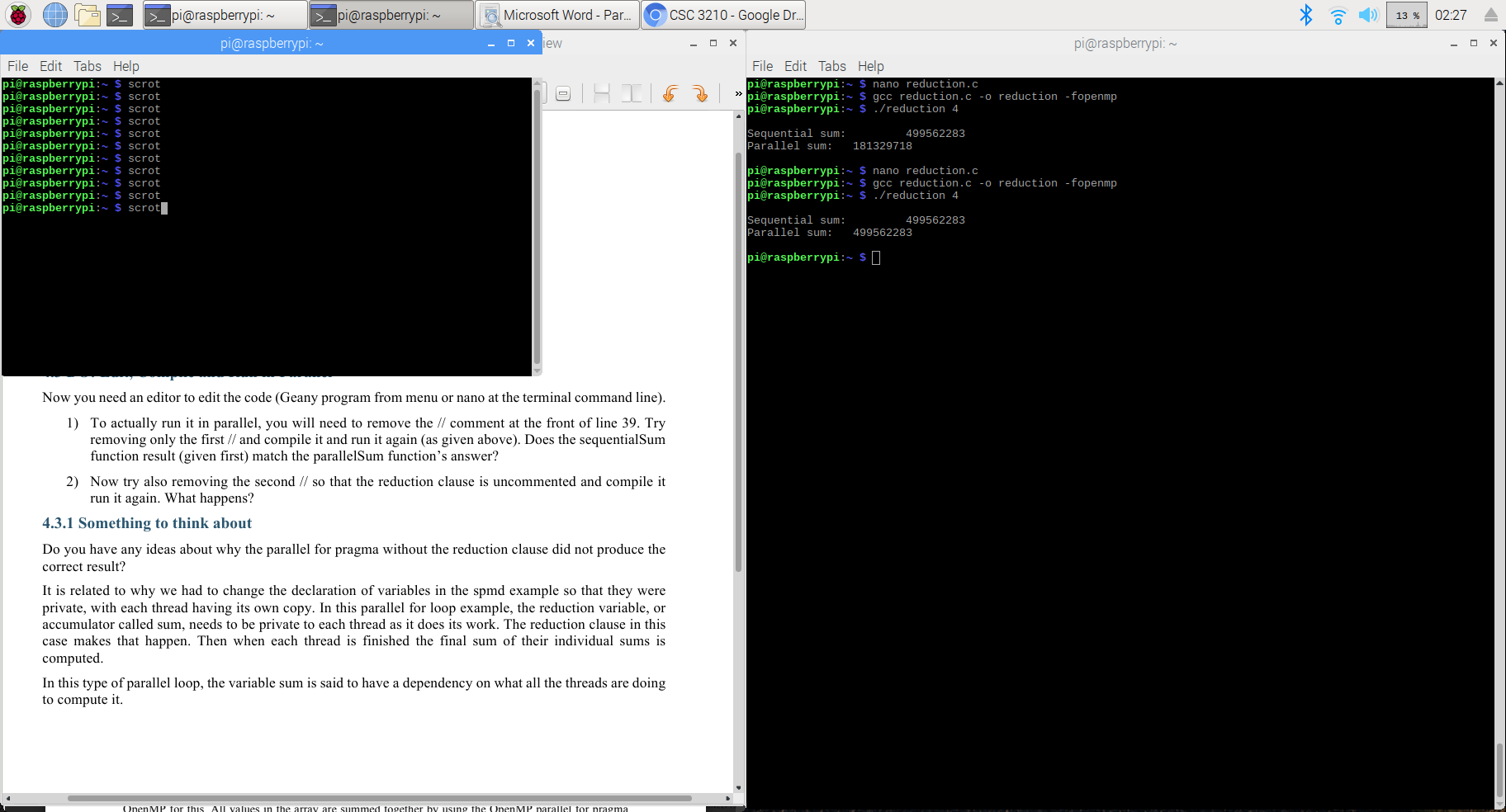
* nano reduction.c
* Note that “#pragma omp parallel for reduction (+:sum)” is in the comment.
* To run the program in parallel, the comment indicators (//) need to be removed.
* All values in the array are summed together by using the OpenMP parallel for pragma with the reduction(+:sum) clause on the variable sum.
* The plus sign in the pragma reduction clause indicates the variable sum is being computed by adding values together in the loop.
* gcc reduction.c -o reduction -fopenmp

1. ./reduction 4 (4 threads)  
   ./reduction 3 (3 threads)

* Sequential Sum: 499562283
* Parallel Sum: 499562283

1. ./reduction (the number of threads not specified)

* 4 threads by default



* Running parallelSum

1. Removed only the first //

* Sequential sum result (499562283) does not match the Parallel sum result (181329718).

1. Removed the second //

* The reduction clause is uncommented.
* Sequential sum result (499562283) matches the Parallel sum result (499562283).
* Why doesn’t the parallel for pragma without the reduction clause produce the correct result?
* In this parallel for loop example, the reduction variable, or accumulator called sum, needs to be private to each thread as it does its work. The reduction clause in this case makes that happen. When each thread is finished, the final sum of their individual sums is computed.