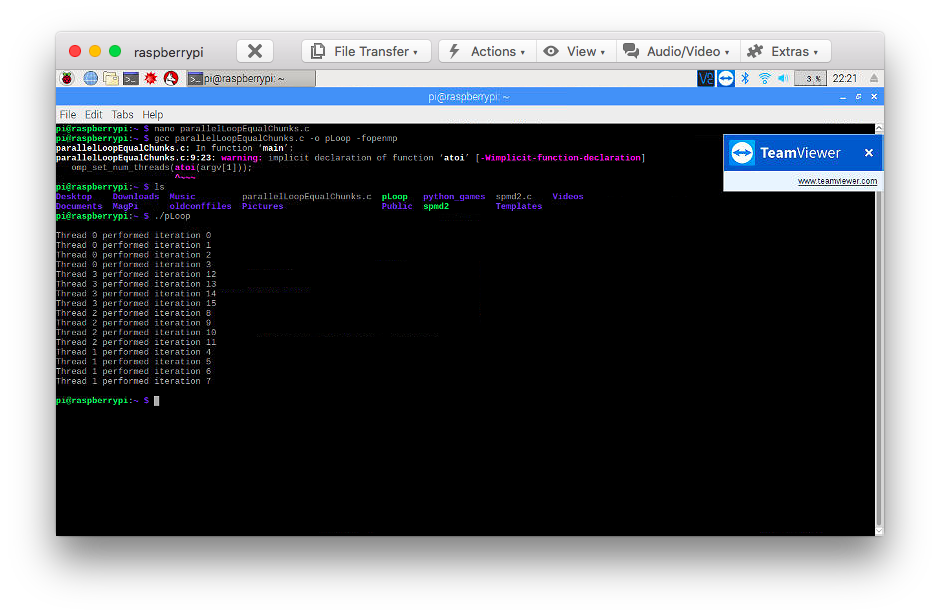
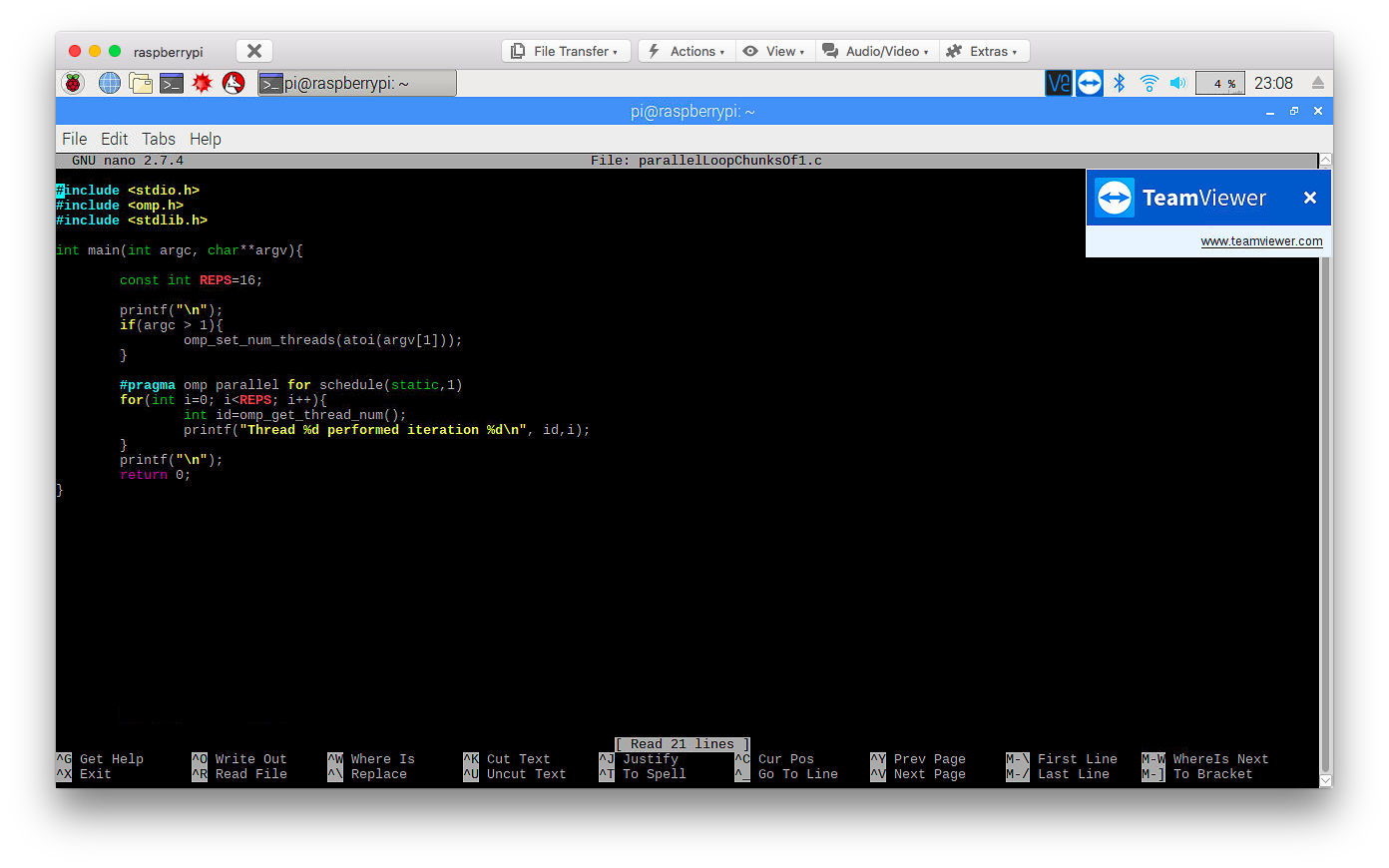


Here we wrote out the first program in terminal using nano editor.

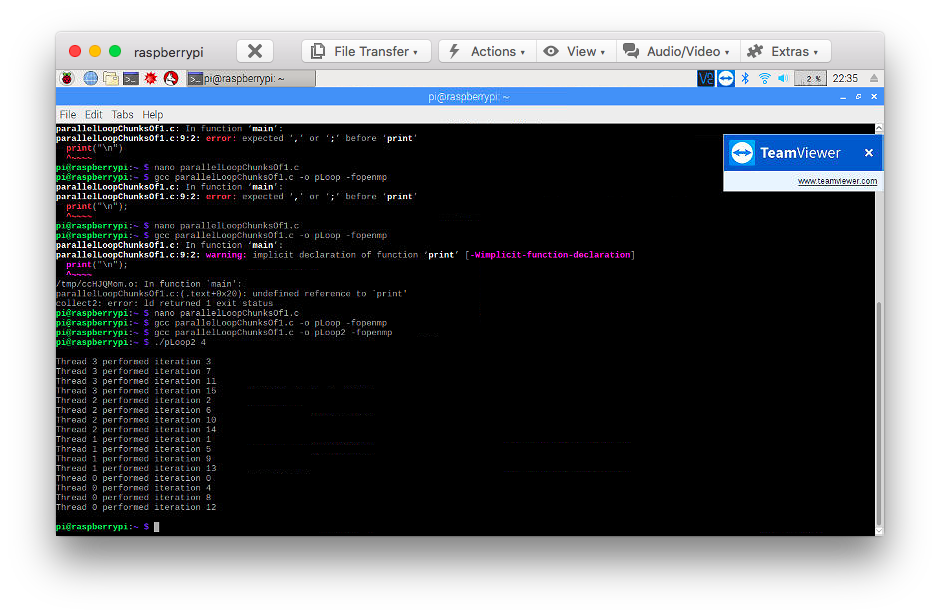
After that we compiled the code using this command **gcc parallelLoopEqualChunks.c -o pLoop -fopenmp**



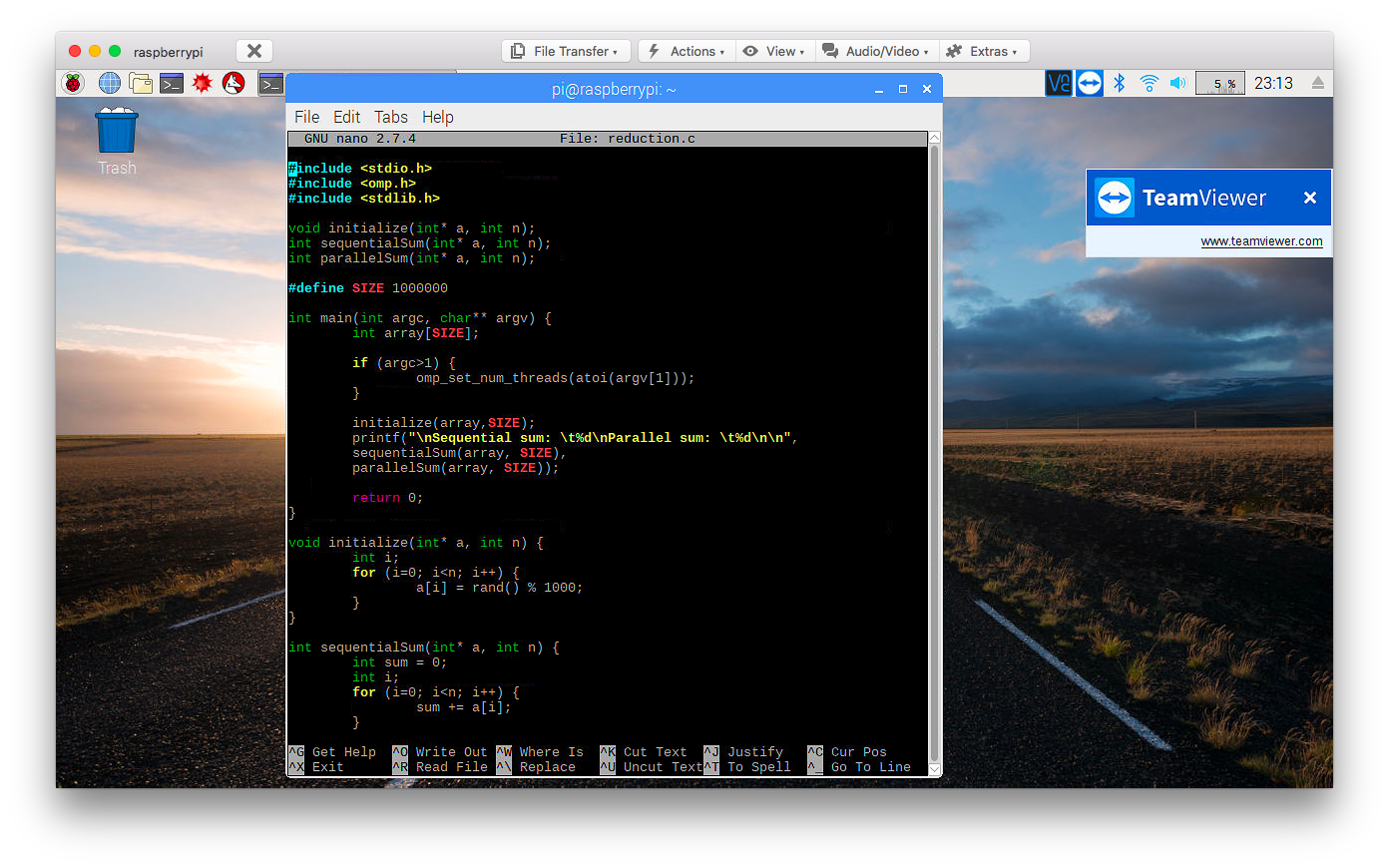
Here we ran the program and found out the threads were running their iterations all at once. Basically when a thread was chosen to start running their iterations, they needed to complete their sequence before moving on to the next thread.

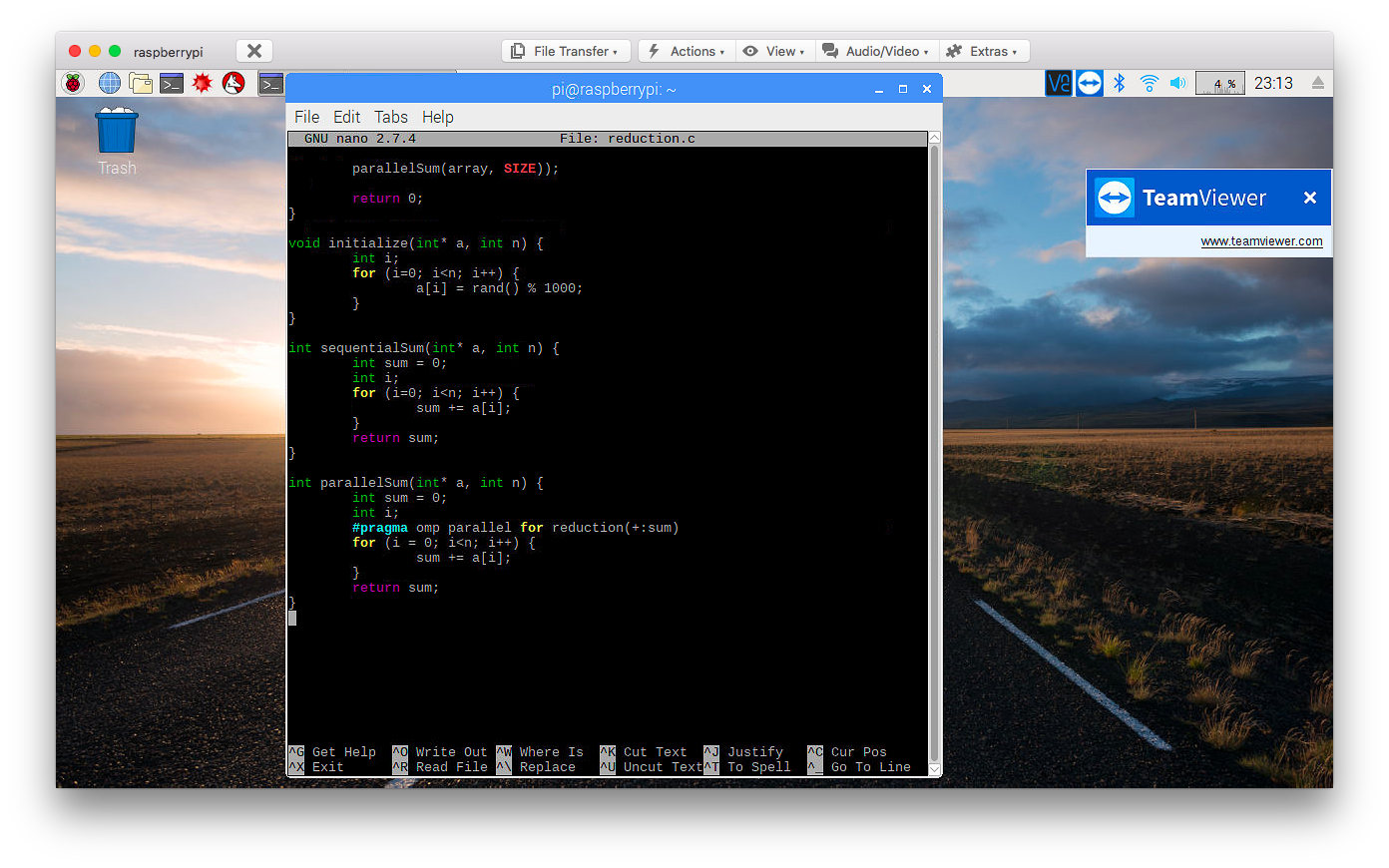


Here we created the second program pLoop2.c. We noticed that a function called schedule was now being used with omp.

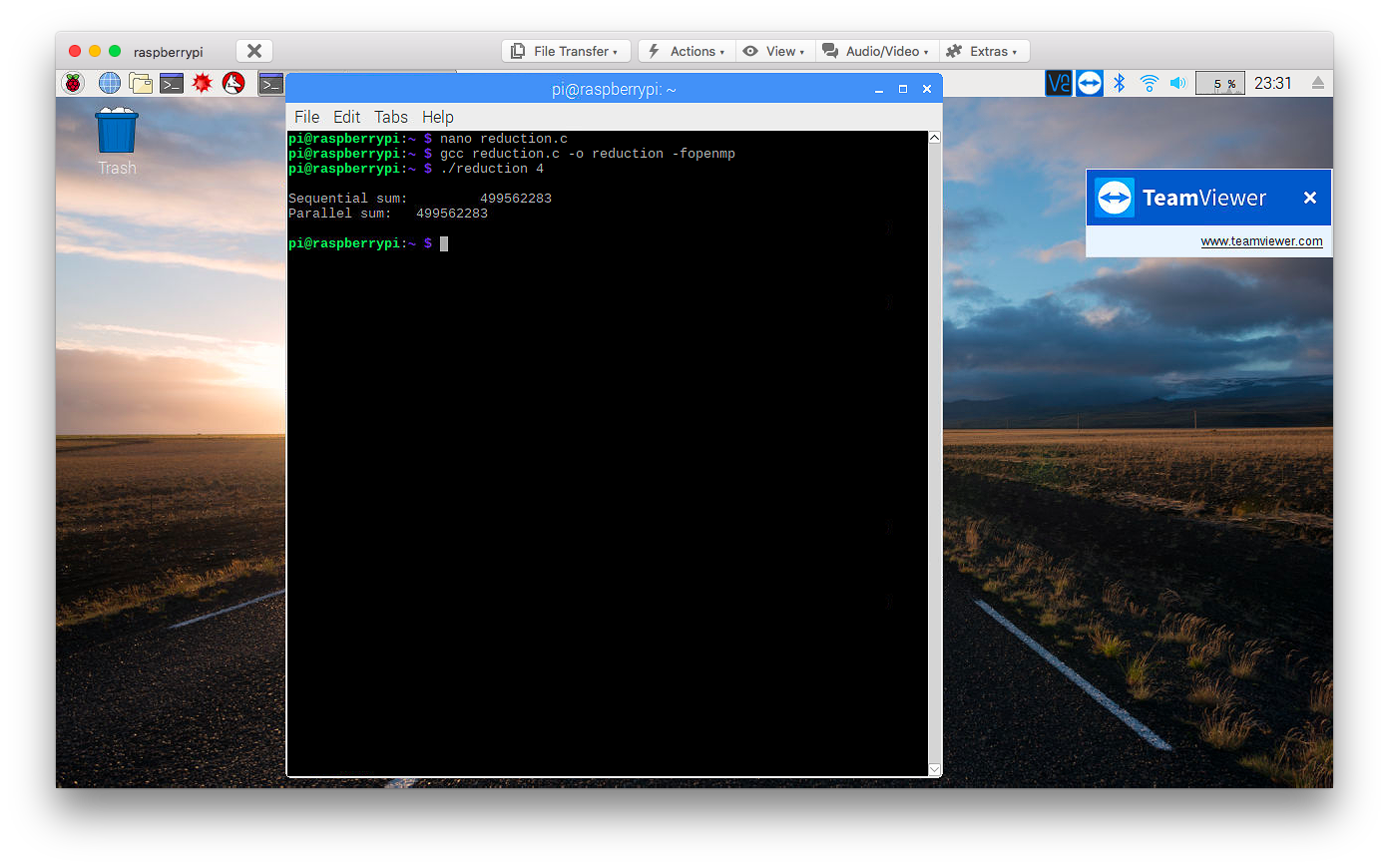


Rather than each thread doing several consecutive iterations of a loop, the second program pLoop2 utilizes static scheduling so that each thread takes turns completing iterations on a regular pattern. For example, thread 0 does iteration 1, thread 1 does iteration 2, etc. Once a thread completes an iteration, it can immediately get the next one assigned to it.



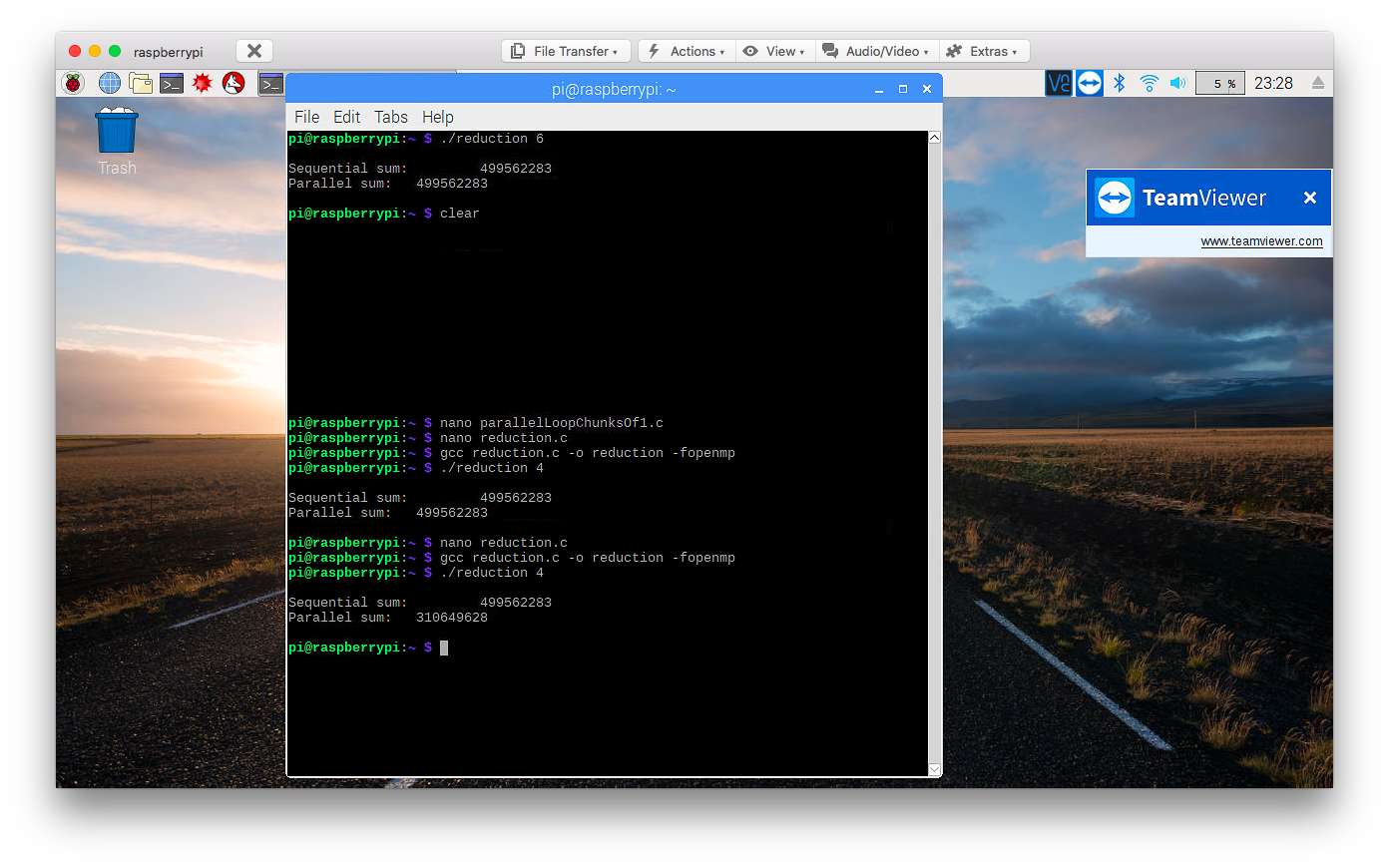


In this third program called reduction, an array is initialized and filled with random integers. Then, 2 different functions are called to calculate the sum of the contents in the array: sequentialSum and parallelSum. The sequentialSum function simply adds up all the values in the array by iterating through it using a for loop.



Here we ran the program for the first time and noticed that both values were the same.

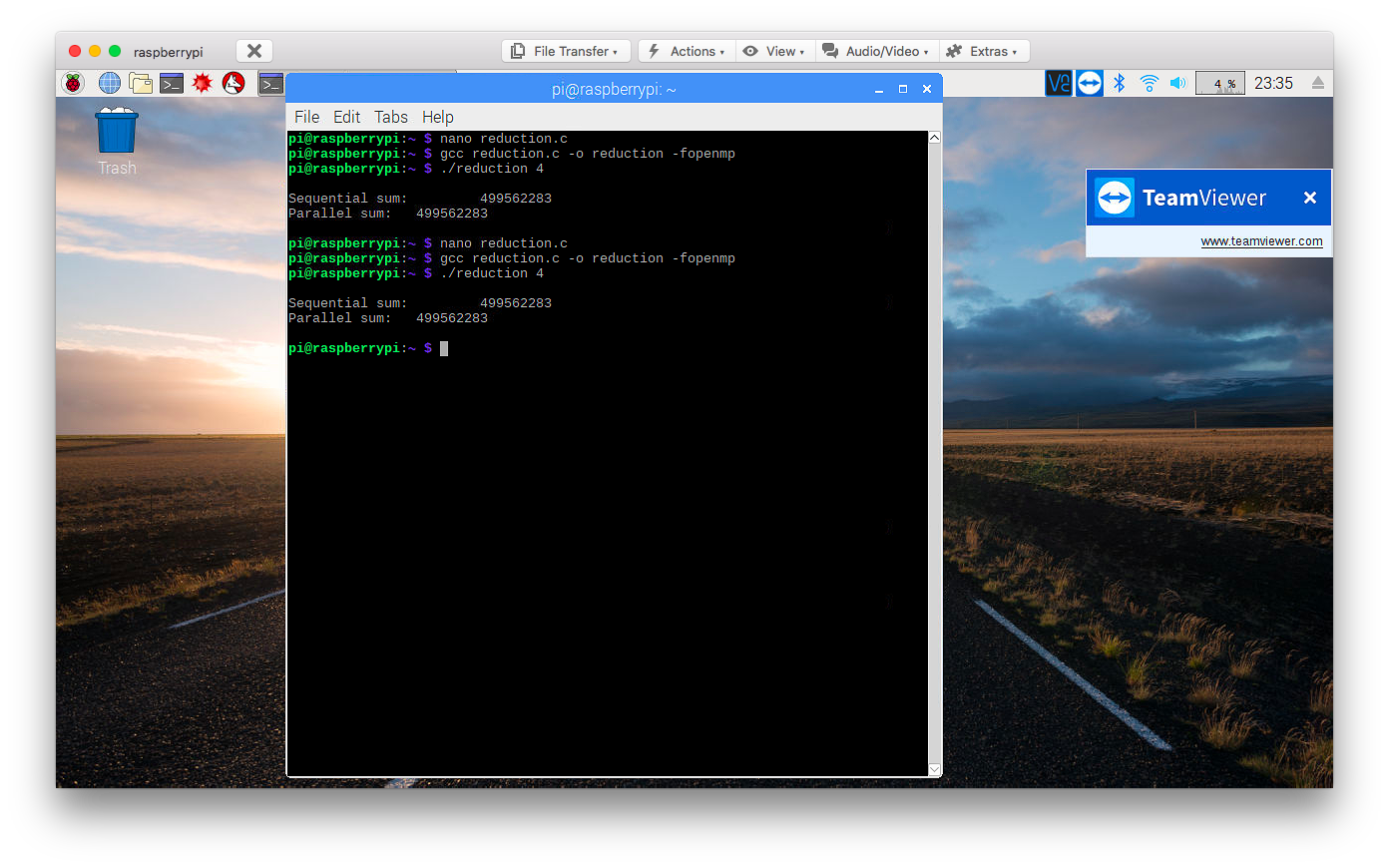
The parallelSum accomplishes the same thing but faster, by running the for loop in parallel with different threads.



Here we ran the code after removing the first comment from the openmp declaration.

However, a problem now arises in that the threads need a way to update the overall sum while each one is working on a different portion of the array. Since there are different threads involved sum is unable to compute the values from the other threads running concurrently. Because of this, the parallel sum outputs a different value than the sequential sum.

To fix this issue, the reduction(+:sum) clause is necessary. This indicates to compute the sum variable by adding up all the values together in the for loop.



Here we ran the code after remove the comment from the reduction function.

We hypothesized that the reason the parallelSum function did not work properly without the reduction clause was because there was no way for the separate threads to know to combine all of their values together into one single sum variable. We think that each thread actually has its own sum variable as they iterate through the data. For example, if the code forked into 4 threads, then there would by 4 different sum variables within each thread.