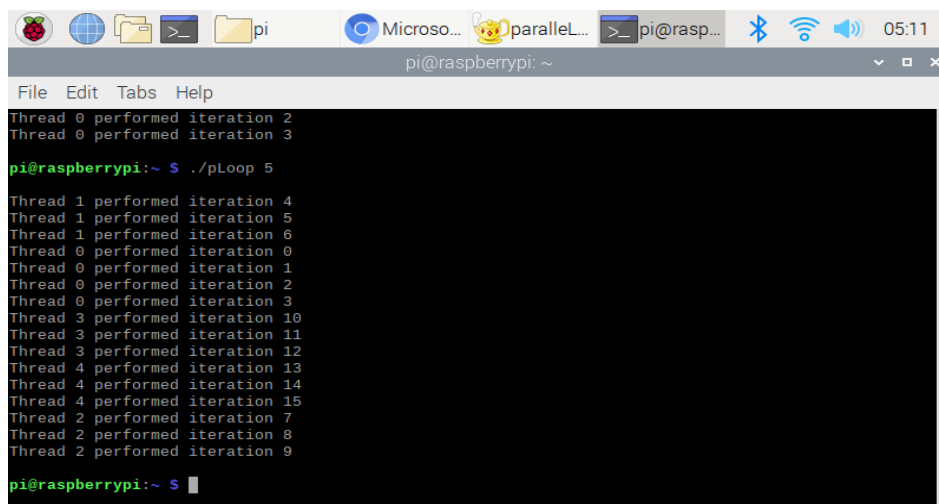


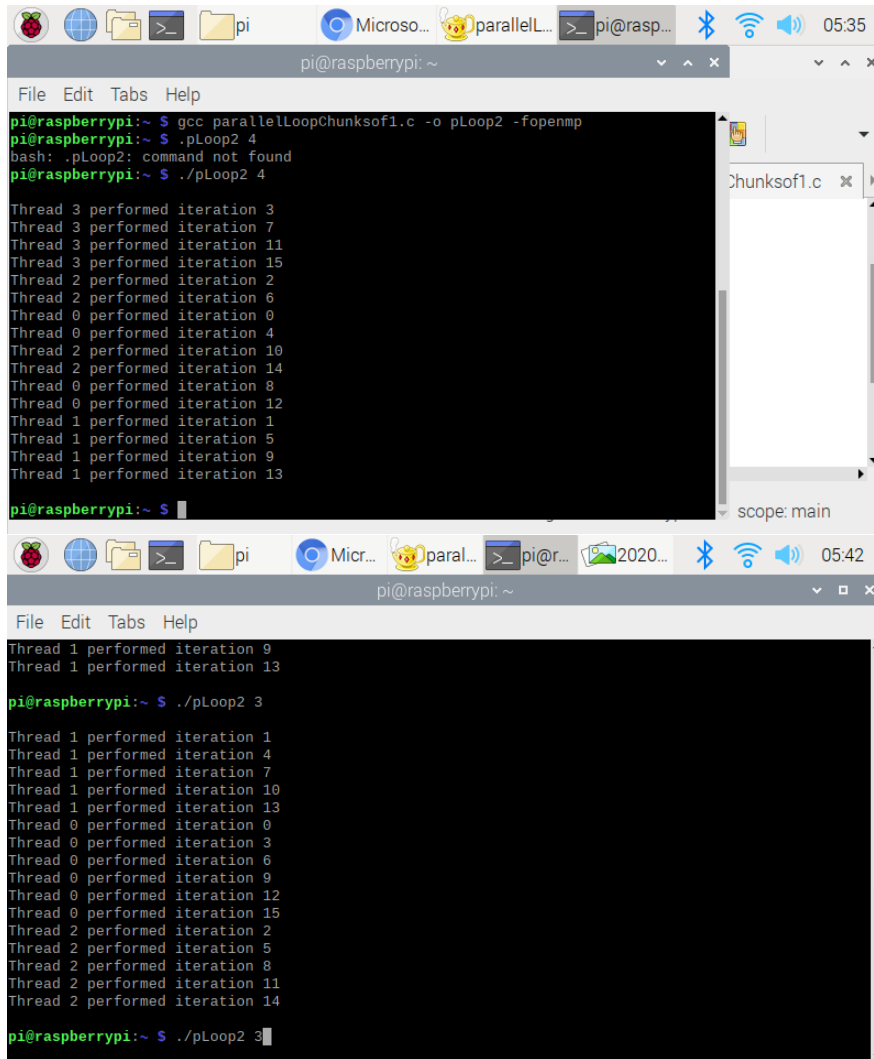
```
pi@raspberrypi: ~  
File Edit Tabs Help  
pi@raspberrypi:~ $ ./pLoop 4  
bash: ./pLoop: No such file or directory  
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp  
pi@raspberrypi:~ $ ./pLoop 4  
Thread 0 performed iteration 0  
Thread 3 performed iteration 12  
Thread 2 performed iteration 8  
Thread 1 performed iteration 4  
Thread 1 performed iteration 5  
Thread 1 performed iteration 6  
Thread 3 performed iteration 13  
Thread 2 performed iteration 9  
Thread 2 performed iteration 10  
Thread 2 performed iteration 11  
Thread 3 performed iteration 14  
Thread 3 performed iteration 15  
Thread 1 performed iteration 7  
Thread 0 performed iteration 1  
Thread 0 performed iteration 2  
Thread 0 performed iteration 3  
pi@raspberrypi:~ $ ./pLoop 4
```



```
pi@raspberrypi: ~  
File Edit Tabs Help  
Thread 0 performed iteration 2  
Thread 0 performed iteration 3  
pi@raspberrypi:~ $ ./pLoop 5  
Thread 1 performed iteration 4  
Thread 1 performed iteration 5  
Thread 1 performed iteration 6  
Thread 0 performed iteration 0  
Thread 0 performed iteration 1  
Thread 0 performed iteration 2  
Thread 0 performed iteration 3  
Thread 3 performed iteration 10  
Thread 3 performed iteration 11  
Thread 3 performed iteration 12  
Thread 4 performed iteration 13  
Thread 4 performed iteration 14  
Thread 4 performed iteration 15  
Thread 2 performed iteration 7  
Thread 2 performed iteration 8  
Thread 2 performed iteration 9  
pi@raspberrypi:~ $
```

The above screenshot is the output for the `parallelLoopEqualChunks` in which the CPU decomposed the tasks or the amount of iterations (16) evenly to the specified fork size through the command-line argument (4.) So, each thread did 4 iterations as you can see. The second screenshot is the output when I changed the number of threads to odd number 5. This time we have 16 iterations, which is not evenly divisible by 5 so the CPU assigned the first thread '0' in this case to do one more iteration than others because all the iterations has to be performed to display the output for the user.

The bottom two screenshots are the output I got from the static scheduling data decomposition program (parallelLoopChunksof1). The first screenshot is for the 4 forked thread size and the second one is for 3 forked thread size. The difference between this program and the parallelLoopChunks program is, in this program the iterations that are done by each thread are not sequential as you can see because of the static clause to the omp parallel for pragma we specified to it. For example, thread 3 performed iteration 3 then it jumped and performed iteration 7 and so on. When the number of threads forked to three the threads still performed unordered iterations but the first thread or thread zero performed 2 more iterations than the other three threads in order to complete all the 16 iterations.



The image contains two screenshots of a terminal window on a Raspberry Pi. The top screenshot shows the output of a program with 4 threads, and the bottom screenshot shows the output of a program with 3 threads. Both outputs show the sequence of iterations performed by each thread, demonstrating non-sequential execution due to static scheduling.

**Top Screenshot (4 threads):**

```

pi@raspberrypi:~ $ gcc parallelLoopChunksof1.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ ./pLoop2 4
bash: ./pLoop2: command not found
pi@raspberrypi:~ $ ./pLoop2 4

Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 2 performed iteration 10
Thread 2 performed iteration 14
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
pi@raspberrypi:~ $

```

**Bottom Screenshot (3 threads):**

```

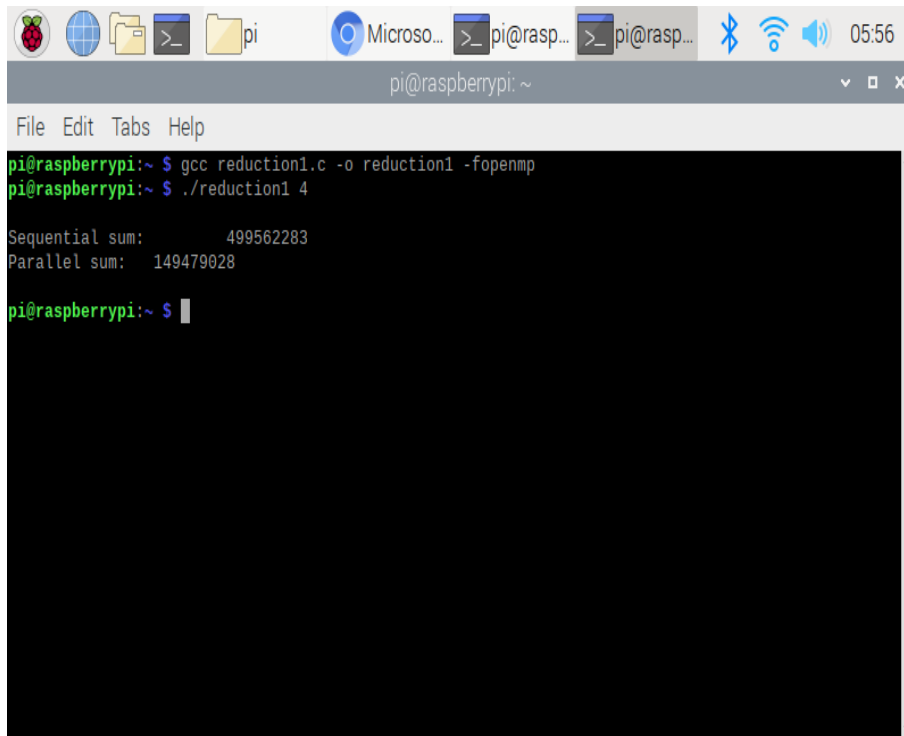
pi@raspberrypi:~ $ ./pLoop2 3

Thread 1 performed iteration 9
Thread 1 performed iteration 13
pi@raspberrypi:~ $ ./pLoop2 3

Thread 1 performed iteration 1
Thread 1 performed iteration 4
Thread 1 performed iteration 7
Thread 1 performed iteration 10
Thread 1 performed iteration 13
Thread 0 performed iteration 0
Thread 0 performed iteration 3
Thread 0 performed iteration 6
Thread 0 performed iteration 9
Thread 0 performed iteration 12
Thread 0 performed iteration 15
Thread 2 performed iteration 2
Thread 2 performed iteration 5
Thread 2 performed iteration 8
Thread 2 performed iteration 11
Thread 2 performed iteration 14
pi@raspberrypi:~ $ ./pLoop2 3

```

The bottom screen shots are the result I got after running the reduction.c program without the changes as it is, with uncommenting the front of line 39 and uncommenting reduction clause. As you can see, I got the same numbers for both sequential and partial sum and even if I changed the thread size to different numbers, the numbers are still the same.

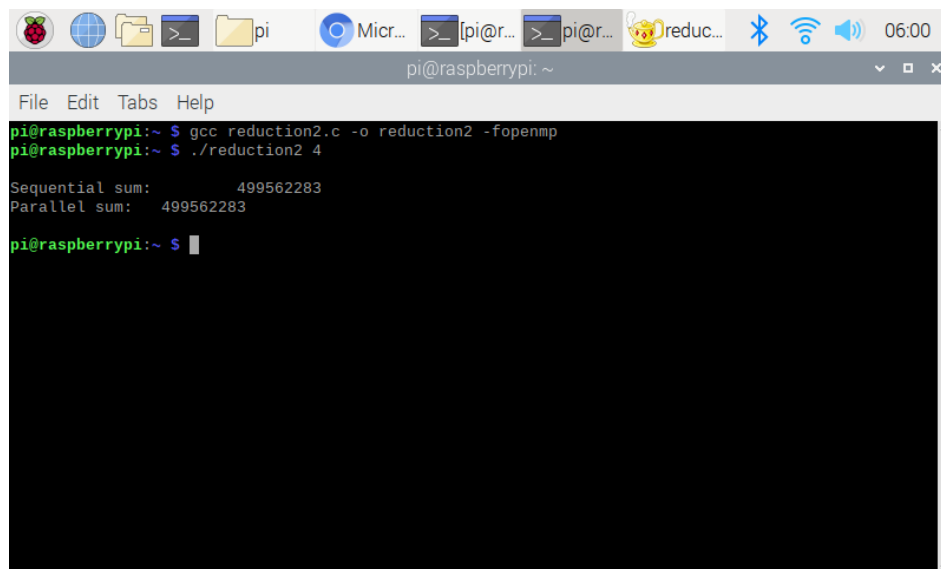


```
pi@raspberrypi:~$ gcc reduction1.c -o reduction1 -fopenmp
pi@raspberrypi:~$ ./reduction1 4

Sequential sum:      499562283
Parallel sum:    149479028

pi@raspberrypi:~$
```

When I uncomment the front of line 39 and run the program, I got different numbers for sequential sum and parallel sum as you can see in the above screenshot. The reason is the sum accumulator haven't been privatized for each threads because the reduction clause, who are responsible to do that is commented so the sum accumulator displays the result of sequential sum from each thread but can't wait until parallel computations performed or their individual sum calculated.



```
pi@raspberrypi:~$ gcc reduction2.c -o reduction2 -fopenmp
pi@raspberrypi:~$ ./reduction2 4

Sequential sum:      499562283
Parallel sum:    499562283

pi@raspberrypi:~$
```

The above screenshot is the output from uncommenting the whole line 39 and running it. These times the numbers are the same for both sequential and partial sum as the accumulator 'the sum' privatized to each threads and wait until each threads complete their individual sum then adds the result to the accumulated value so we get the same result.