**A screenshot of a cell phone

Description automatically generated**

**A screenshot of a cell phone

Description automatically generated**

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.

**A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated**

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. **A screenshot of a cell phone

Description automatically generated**WhenI 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 accumulatoer haven’t been privatized for each threads because the reduction clause, who are resposnible to do that is commmented so the sum accumulatoer displays the result of sequantial sum from each thread but can’t wait until parallel computations performed or their individual sum calculated.

**A screenshot of a cell phone

Description automatically generated**

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.