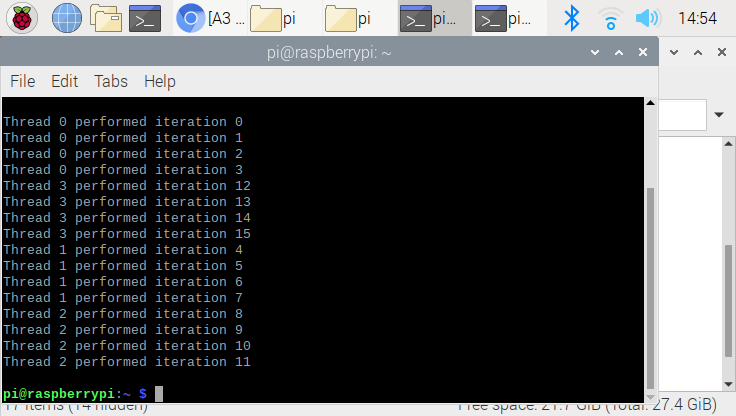
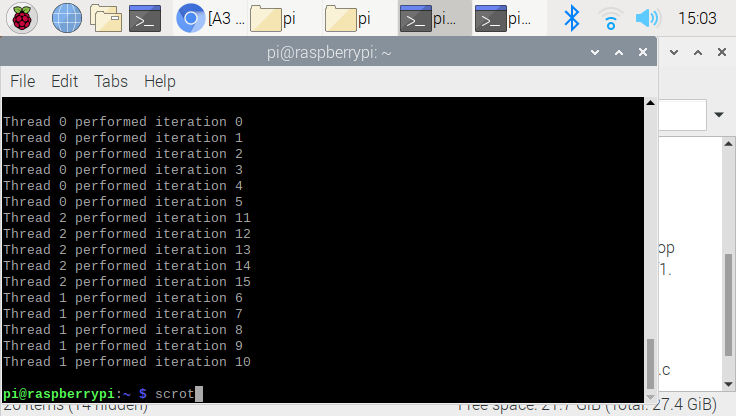
Andy Lee : Parallel Programming Documentation.

Equal Chunk Observation



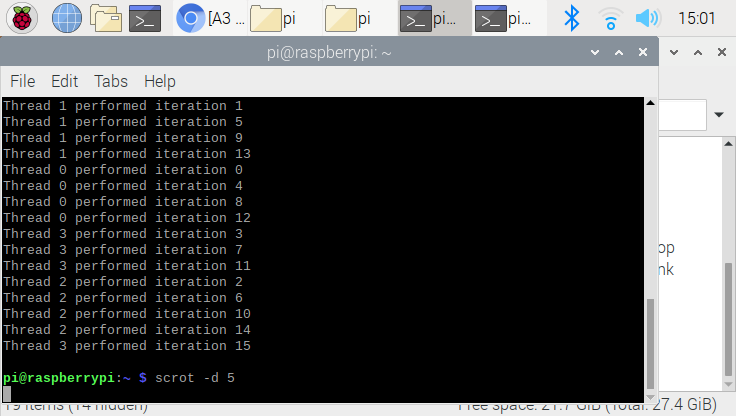
This is a equal chunk result. This program was ran with 4 threads. As the picture shows, each threads distributed their work equally. You could also notice that the iteration goes sequentially from0 to 15. The unique parts about this task is that the iteration always start with 0 threads, and increase by order. We could derive the conclusion from the observation that the threads are communicating each other while they are performing.

**Running with 3 threads**

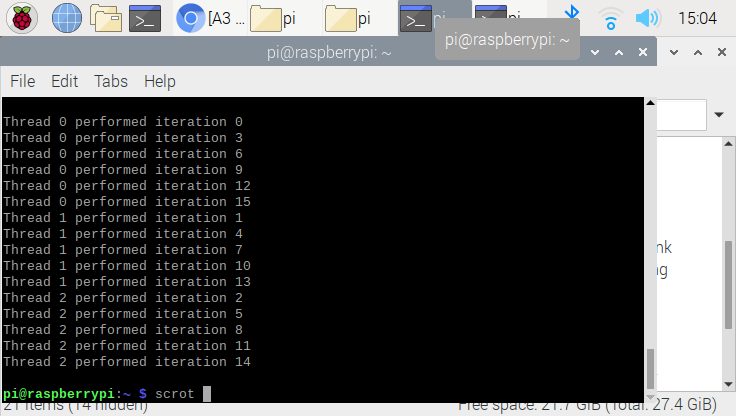


When we run the program with 3 threads, each threads done more tasks than 4 threads. When we declare the 4 threads, each threads are responsible to perform 4 tasks, but now thread 0 done 6 iterations, and others done 5 iteration. We could derive the conclusion from observation that , when we declare the odd number of threads, very first thread(thread 0) would done 1 more task compare to the other threads.

Chunk of 1 Program Observation

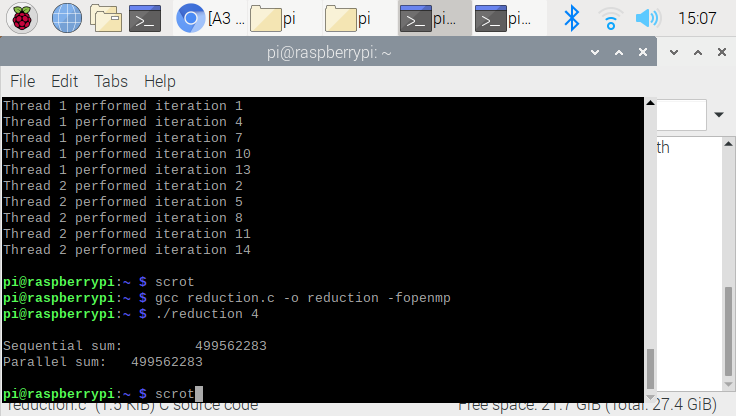


Unlike EqaulChunk, the performance for each threads are not sequential. Instead, the threads are running one by one, (such that thread 0 done iteration 0 , and thread 1 done iteration 1…). One similarity with EqaulChunk is that the threads are equally distributed to their work. We could conclude the result from observation by saying that ; even though it run differently, the program done exactly same task with EqualChunk at the end.

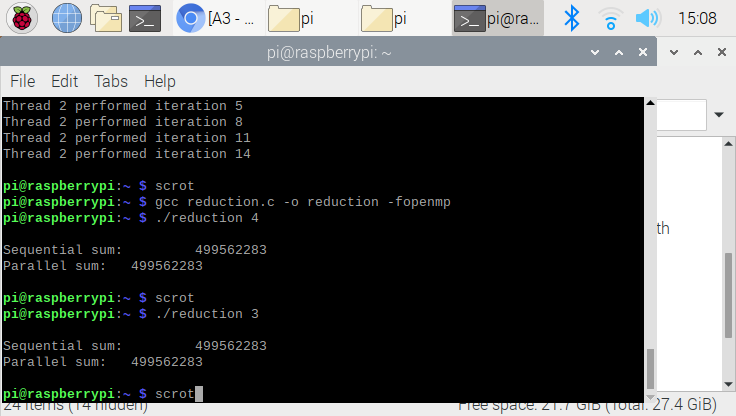


This is program performed with 3 threads. Equal explanation with above where, the threads run their iteration in order corresponds to their threads. (such that thread 0 run first, thread 1 run second ….). Since we only have 3 threads, each threads are responsible to do more tasks compare to the 4threads. Despite the difference number of threads, the task was accomplished same at the end.

**Reduction Observation**

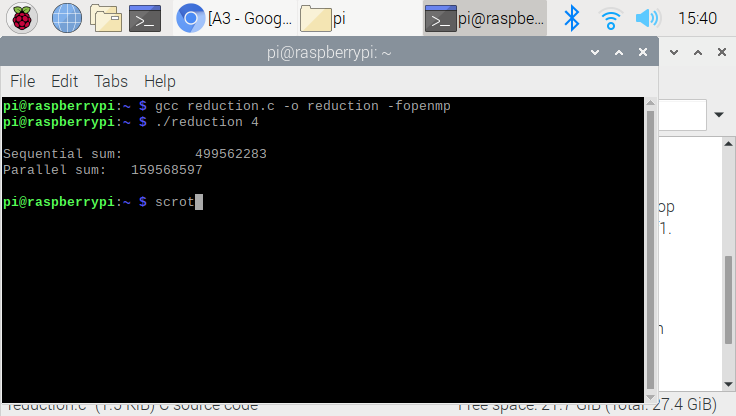


In this program run without comment out, without “pragma” the result is always same, no matter how many threads you are using. This is because the program only sum the array sequentially, not summing the array using the multiple threads. This mean, only one threads would work to sum the array, despite putting “4” in the input section. As you look at the result, the this is reason why Sequential sum and Parallel Sum is same number.

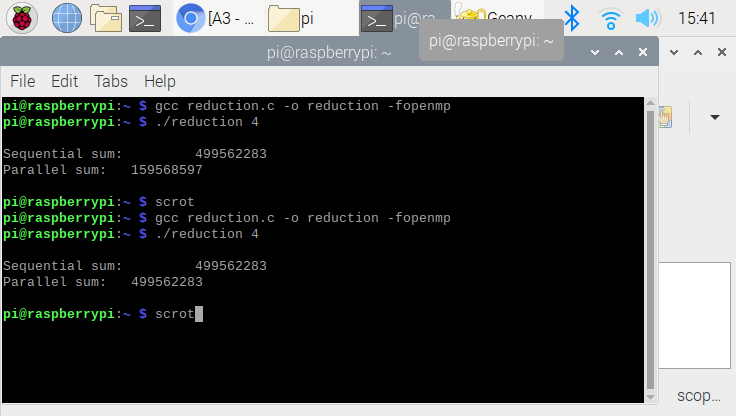


The result is same despite summing with 3 threads. Again, this is because the program only sum the array sequentially. The number is same no matter what input we put.

**Reduction observation: remove comment**



Above result was perform by Remove first // comment. The number for parallel sum was changed due to running a mutli core. Overall, The sequential sum and Parallel sum does not matches.



Once we remove all comment(remove second comment too), the result go back to previous stage, where the result for the sequential sum and parallel sum matches. This is because the operation (+sum), where even though multiple threads was run, they communicate each other, and sum at the end.