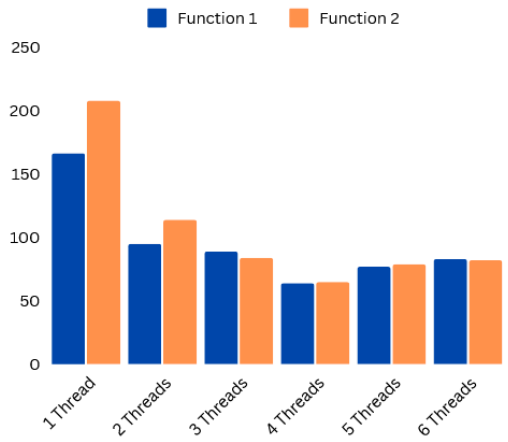
Times (in seconds)

	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads
Funct 1	166.421s	95.231s	89.756s	64.402s	77.357s	83.822s
Funct 2	208.803s	114.476s	84.313s	65.661s	79.057s	82.741s

Speed-up

	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads
Funct 1	1	1.748	1.854	2.584	2.151	1.985
Funct 2	1	1.824	2.477	3.180	2.641	2.524

Execution Time (seconds) vs Thread Count



Results of using no locks with a limit of 2,000,000,000

	Run 1	Run 2	Run 3	Run 4	Run 5
Funct 1 Evil	1000010004	999994997	999999993	999995001	999992500
Funct 1 Odi.	1000014997	1000015002	1000002505	999999999	1000004997
Funct 2 Evil	1000014998	999997498	1000002493	1000007499	1000022505
Funct 2 Odi.	1000017505	1000022500	1000012505	1000017503	1000029994

Results and System Information.
When the limit was 1,000,000,000 the count of evil numbers was 499,999,999 and the count of odious numbers was 500,000,001. I ran these programs through an Ubuntu Virtual Machine with base memory of 8,192 MB and 6 processors.

What happens when there are no locks
To test what would happen if there were no spinlocks for the program, I removed the locks in both function 1 and 2 that were used to access the current index as well as update the global counters. This changed the final count of the evil and odious numbers up to a change of 20,000 in the 5 times the program ran for each function.

Difference between the two functions
Both function 1 and 2 come to the same conclusion with the only difference being when they update the global count variables. Function 1 updates the global count variables in a spinlock every time an evil number or odious number is found. Function 2, on the other hand, updates the global count variables in a spinlock at the end of the block. This makes it so that function 2 does a lock less often than function 1 does.

Evil/Odious Counter Update Relocation
Since function 2 is only supposed to update the global counters once every block while function 1 is supposed to update the global counters every time an evil or odious number was found, the locks had to happen in different parts of the function. Function 1's lock happens inside a for-loop that iterates over the block. Because the lock is inside the for-loop, the lock is used every iteration of the for-loop. Function 2, on the other hand, has its lock immediately following the for-loop. This makes it so that the lock is only called once for every time the for-loop finishes running, rather than every iteration of the for-loop.