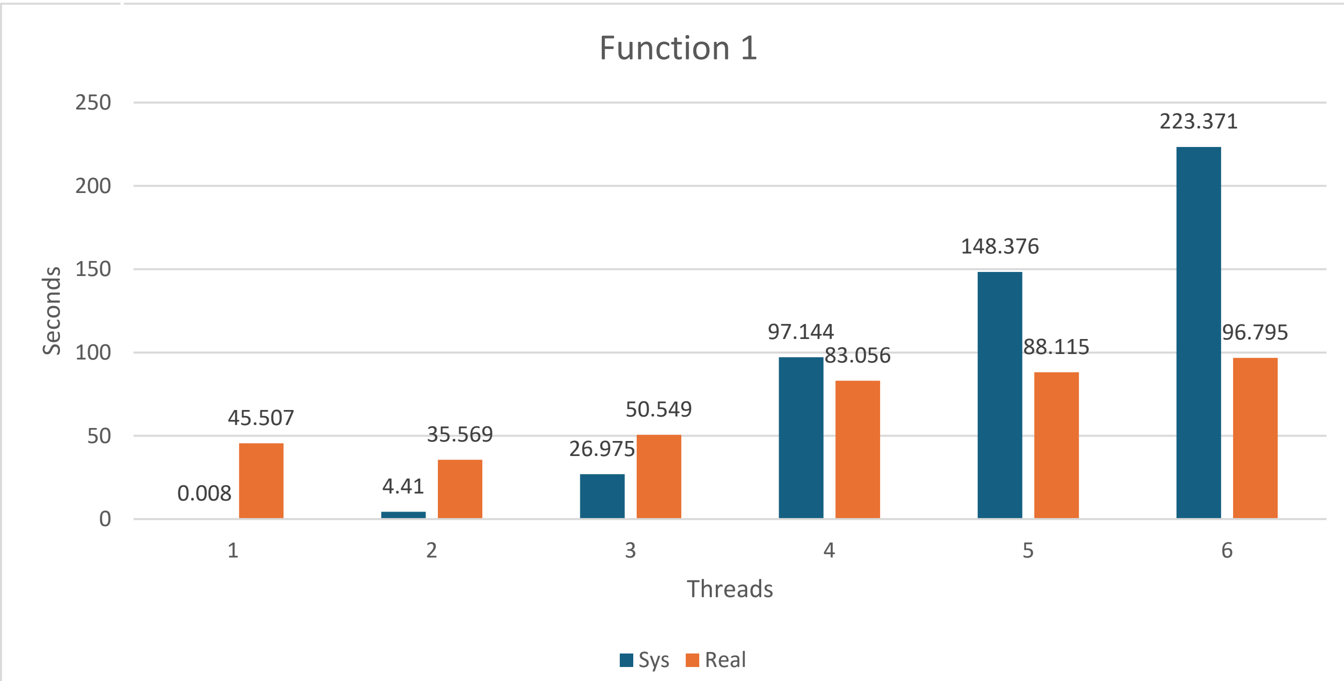
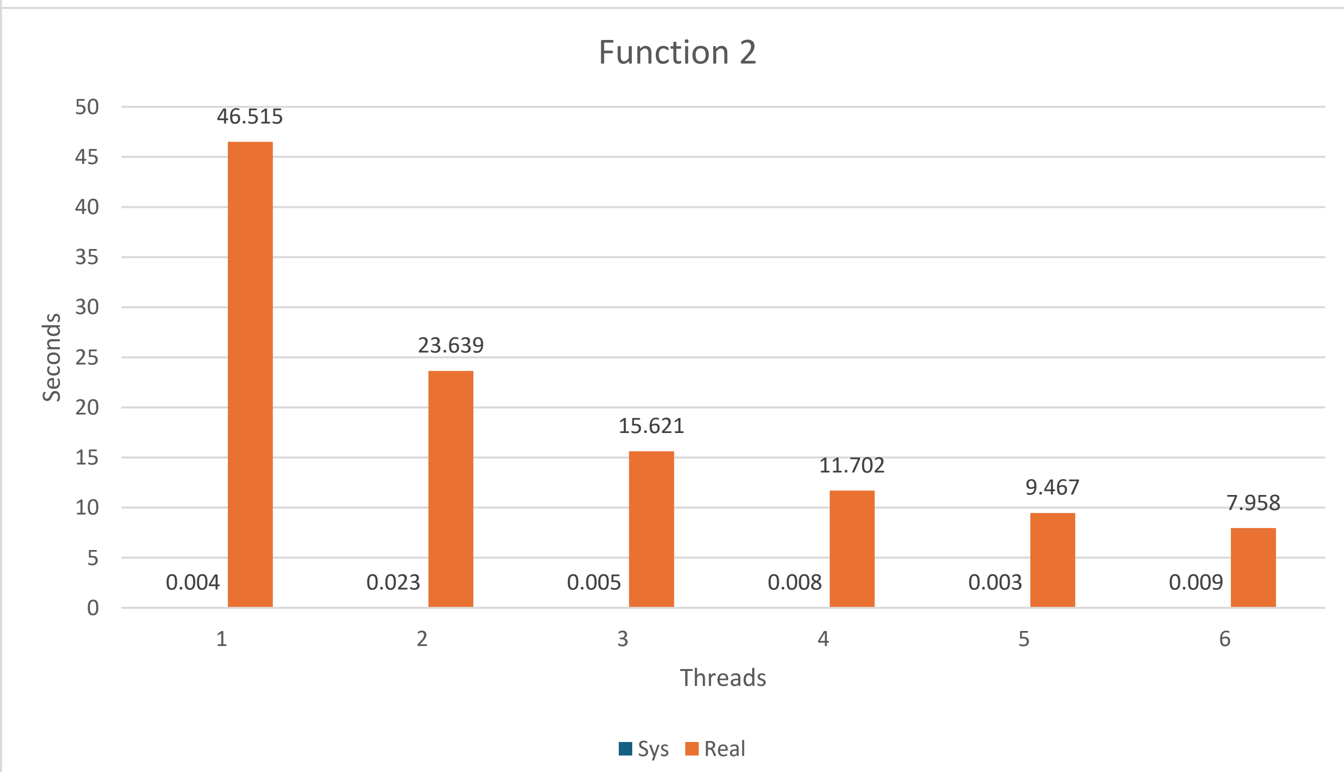
Harrison Atherton
Professor Ed Jorgensen
CS 370

Project 5a – Threads





This program was executed on a virtual machine with the following specifications.
Processors: 14 (13^{th} Gen Intel i5-13600KF)
Approximately 12GB of DDR4 RAM
Motherboard: Z690 PG Riptide

Summary

Looking at function 1 first, we notice that the more threads we add to the program, the slower it becomes. This is most likely since each time a number is found, it will create a lock to update the global counter. If there are many threads working at the same time, they will each fight for that lock, which will cause delays and slow the program down. Frequently locking and unlocking a program will lead to slower results. There is also the possibility of the cache misses, since we are accessing a global variable, it may not be in the cache at that very moment.

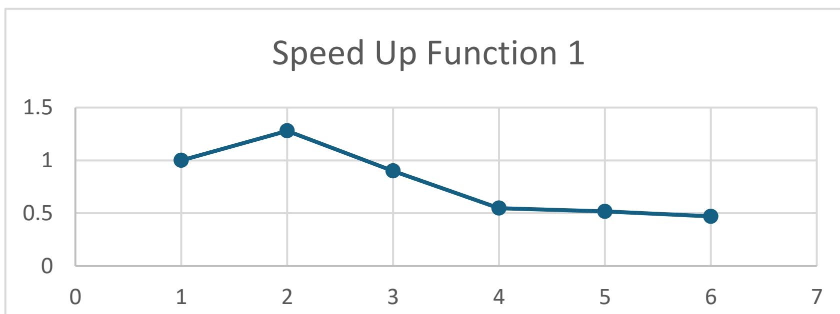
It's also important to notice that the system time increases significantly within function 1. This is most likely due to the amount of context switching we are doing since we are updating the count and acquiring a lock every time.

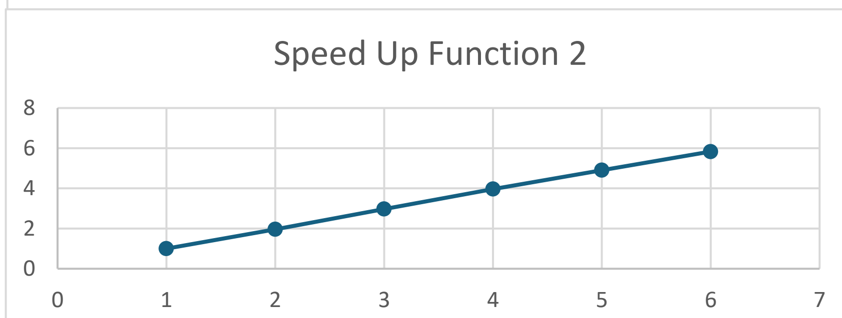
Looking at function 2, we notice that the overall real time is significantly reduced when updating the local variables first, acquiring a lock for the global variables, and then incrementing the global variables by the local variables. This is an efficient usage of locks, as it reduces context switches and improves performance.

Compared to function 1, it is noticeable that the system time is significantly less. This is due to acquiring a lock a lot less often.

Threads	Speed Up
1	1
2	1.279400602
3	0.900255198
4	0.547907436
5	0.516450094
6	0.47013792

Threads	Speed Up
1	1
2	1.963450231
3	2.971256642
4	3.966330542
5	4.902714693
6	5.832369942





Speed-Up

As we can see by the speed up charts, function 1 gets a minimal speed up when using two cores. This is likely because there are a lot less locks being acquired, and it *luckily* managed to work efficiently. However, when adding more threads, we see that it seems to go down in a random fashion.

When looking at function 2, the speed up is quite linear overall. This shows that the more threads that we add to this program, it will likely stay in this linear fashion, resulting in faster execution times.

Function 1	Function 2
Evil/Odious Numbers Results 1 Evil Number Count: 560445525 Odious Number Count: 568248774 real 0m39.370s user 3m56.167s sys 0m0.021s **Evil/Odious Numbers Results 2** Evil Number Count: 565039207 Odious Number Count: 567913750 real 0m38.907s user 3m53.416s sys 0m0.009s **Evil/Odious Numbers Results 3** Evil Number Count: 641382421 Odious Number Count: 643298843 real 0m38.175s user 3m48.986s sys 0m0.018s **Evil/Odious Numbers Results 4** Evil Number Count: 585752007 Odious Number Count: 588903819 real 0m38.974s user 3m53.760s sys 0m0.016s **Evil/Odious Numbers Results 5** Evil Number Count: 617040314 Odious Number Count: 631198443 real 0m39.800s user 3m58.733s sys 0m0.021s	**Evil/Odious Numbers Results 1** Evil Number Count: 999930007 Odious Number Count: 999909991 real 0m13.714s user 1m22.252s sys 0m0.010s **Evil/Odious Numbers Results 2** Evil Number Count: 999962495 Odious Number Count: 999940007 real 0m13.790s user 1m22.704s sys 0m0.011s **Evil/Odious Numbers Results 3** Evil Number Count: 999975016 Odious Number Count: 999954983 real 0m13.719s user 1m22.291s sys 0m0.004s **Evil/Odious Numbers Results 4** Evil Number Count: 999984990 Odious Number Count: 999960011 real 0m13.643s user 1m21.827s sys 0m0.006s **Evil/Odious Numbers Results 5** Evil Number Count: 999982501 Odious Number Count: 999964999 real 0m13.638s user 1m21.799s sys 0m0.011s

Lock Removal

Analyzing the function with no locks at a 2 million limit, we can see that the Evil and Odious counts are off. This is because there is a race condition occurring, where two threads access the same data concurrently. This results in incorrect data.

Function 1 results are significantly off. Function 2's results are closer to the expected counts, but still incorrect. This improvement likely comes from updating counts in a local variable first and then updating the global counters at the end. This possibly reduces the chance of race conditions by minimizing how often threads interact with the global counter.

Evil/Odious Counter Update Relocation

When the Evil/Odious number counters were moved to be updated only after each thread completed counting for its assigned block, the performance and accuracy of the program improved significantly. The performance improved because we are accessing locks a lot less frequency, and the accuracy improved because when we update the counters only after processing each block, the likelihood of race conditions will decrease.