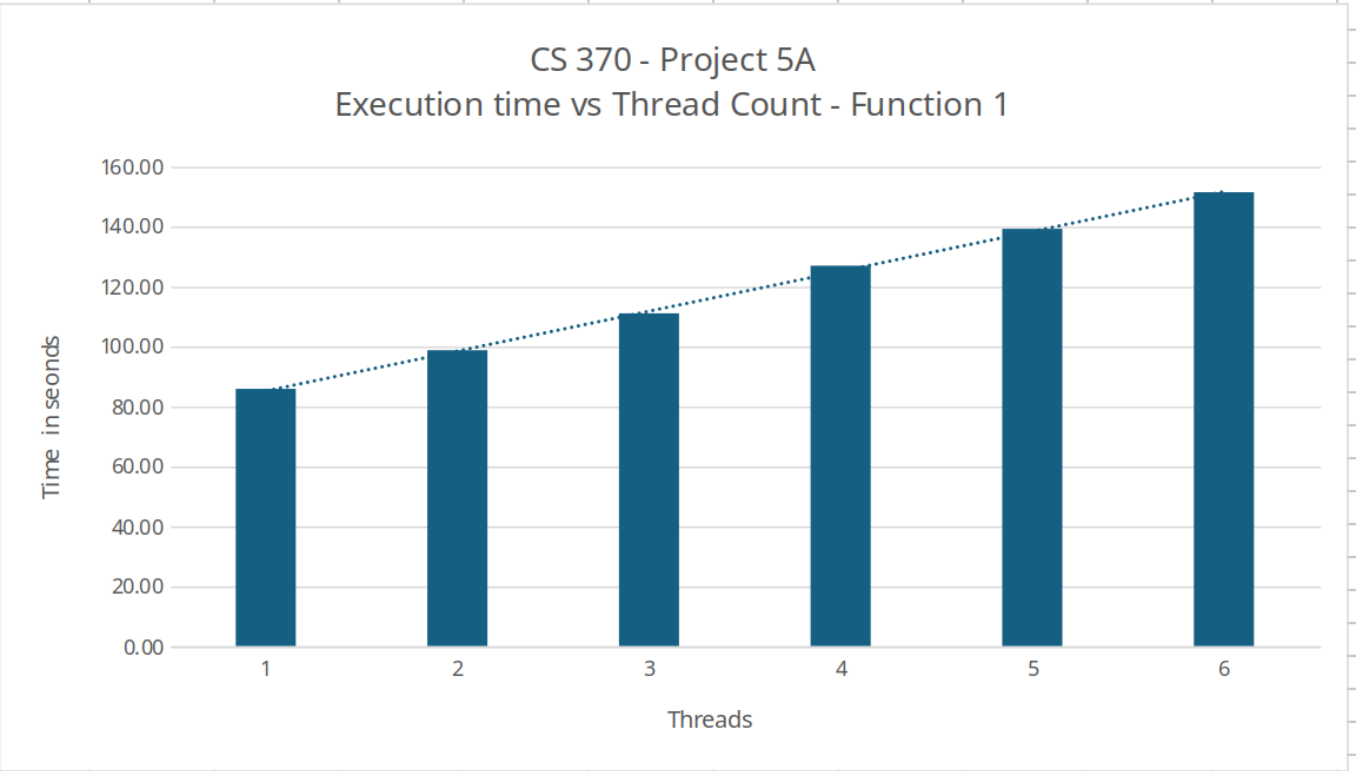
Project 5 Report

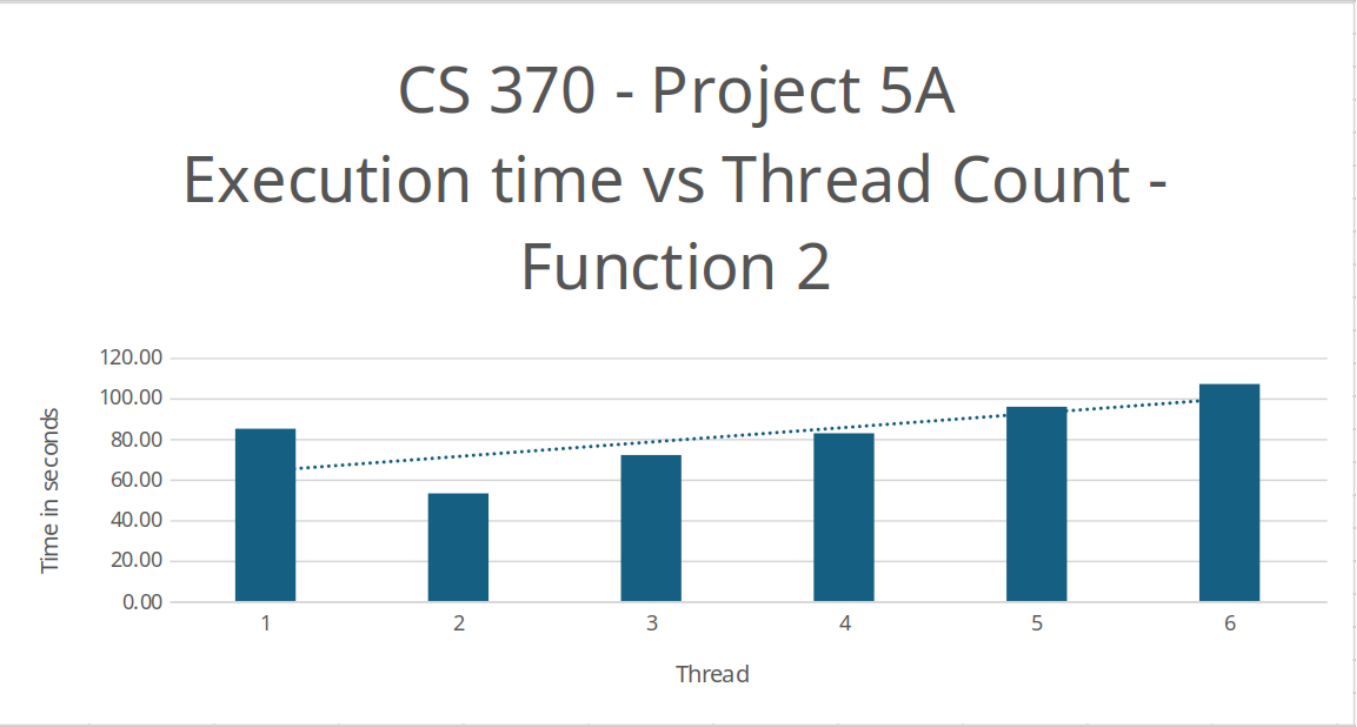
Summary

Machine Summary

- Acer Swift
  - Architecture: x86_64
  - AMD Ryzen 7 6800
  - Cores 8
    - Threads per core 2
  - Cache
    - L1d 256 kib - 8 instance
    - L1i 256 Kib - 8 instance
    - L2 4 MB - 8 instance
    - L3 16 MB - 1 instance

Program Summary





Lock Removal

Function 1:

- Run 1
  - Evil: 1485632970
  - Odious: 1489600725
- Run 2

    - Evil: 1495706803
    - Odious: 1500485741
- Run 3
    - Evil: 1495687463
    - Odious: 149535819
- Run 4
    - Evil: 1500214585
    - Odious 1502566038
- Run 5
    - Evil: 1499671981
    - Odious: 1502740543

Function 2:

- Run 1
    - Evil: 687698468
    - Odious: 687761041
- Run 2
    - Evil: 519009875
    - Odious: 519178727
- Run 3
    - Evil: 344952798
    - Odious: 517929775
- Run 4
    - Evil: 515812942
    - Odious: 515879807
- Run 5
    - Evil: 516049084
    - Odious: 516262800

Write Up

```
In my tests we have noted that "more threads" does not necessarily mean more
per formant. In these tests as we can see in both function one as well as
function two the more cores we add we generally see an increase in time for
completion. While 2 cores in function 2 is the most optimal both follow the
same trend.
```

This can be explained by mutexes. What happens is since the data is locked a thread can only access the data to change it one at time. What this does is sleeps the unused threads, then when a thread is invoked to perform it tasks it automatically invokes a cache miss, and then performs its task.

2 Cores is the most optimum in function 2 because we have 2 tasks we can perform locally that can be shared. So 2 cores can be working simultaneously. However as we increase cores and increase time spent waiting we increase cache misses and decrease performance.

Lock Removal

With the removal of the locks we see that we can now have race conditions where multiple threads can change the data at the same time. This is why there is such discrepancy between the data sets.

Counter placement

Function1: Because of the global placement of the counter. We have to force the threads to access the data individually. This leads to a slow down in performance as the data can only be accessed by 1 thread at a time.

Function2: Because counter placement is done locally within the function. It essentially locks itself from the other threads, and a single thread can focus on running that code. While the overall odious and evil counter still have to be locked to be updated to prevent race conditions this also allows for multiple asynchronous threads to be running code at the same time.