Prabhas Kumra
CS 302 – 1004
Assignment #6

Executions for Harshad Numbers program.

****

Executions for Harshad Numbers Program to 5000000000, 1 threads.

CS 302 - Assignment #6

Hardware Cores: 32
Thread Count: 1
Number Limit: 5000000000

Results:
  Count of Harshad numbers between 1 and 5000000000 is 131304670

Thread took: 2865350 milliseconds

****

Executions for Harshad Numbers Program to 5000000000, 2 threads.

CS 302 - Assignment #6

Hardware Cores: 32
Thread Count: 2
Number Limit: 5000000000

Results:
  Count of Harshad numbers between 1 and 5000000000 is 131304670

Thread took: 1279969 milliseconds

****

Executions for Harshad Numbers Program to 5000000000, 4 threads.

CS 302 - Assignment #6

Hardware Cores: 32
Thread Count: 4
Number Limit: 5000000000

Results:
  Count of Harshad numbers between 1 and 5000000000 is 131304670

Thread took: 631060 milliseconds

****

Executions for Harshad Numbers Program to 5000000000, 8 threads.

CS 302 - Assignment #6

Hardware Cores: 32
Thread Count: 8
Number Limit: 5000000000

Results:
  Count of Harshad numbers between 1 and 5000000000 is 131304670

Thread took: 357848 milliseconds

****

Executions for Harshad Numbers Program to 5000000000, 16 threads.

CS 302 - Assignment #6

Hardware Cores: 32
Thread Count: 16
Number Limit: 5000000000

Results:
  Count of Harshad numbers between 1 and 5000000000 is 131304670

Thread took: 269977 milliseconds

****

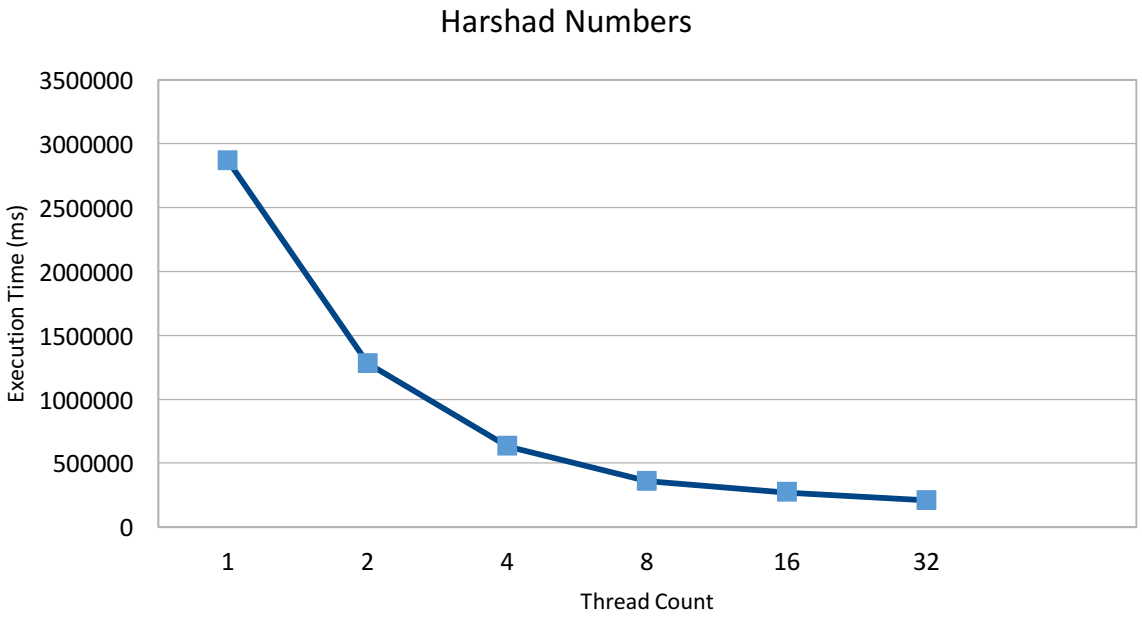
Executions for Harshad Numbers Program to 5000000000, 32 threads.

CS 302 - Assignment #6

Hardware Cores: 32
Thread Count: 32
Number Limit: 5000000000

Results:
Count of Harshad numbers between 1 and 5000000000 is 131304670

Thread took: 208188 milliseconds



The first time I ran the tests with 1 thread, it came out to 2865350 milliseconds. And from then, as the number of cores increased, the execution times cut into half for 2 cores, then again to the half with 4 cores. But as there were further increase in cores, there were not much of time difference between them as there were in the first two times.

Speed up = $(Time_{single}) / (Time_{new})$

- 2 cores = 2865350 / 1279969 = 2.24%
- 4 cores = 2865350 / 631060 = 4.54%

- 8 cores = 2865350 / 357848 = 8.01%
- 16 cores = 2865350 / 269977 = 10.61%
- 32 cores = 2865350 / 208188 = 13.76%

So, when mutex locks were removed, the results for the counts were not consistent, they were different each time the program was ran. The time executions almost remained the same. The results were different because of the race condition. Race condition happened because when multiple threads try to write to a variable at the same time. So when locks were removed, threads tried to increase the counter at the same time and the counter was either not incrementing or was just incrementing once for multiple threads.