Scott Crawford: Scott-Crawford

Sumanyu Gupta: sumanyugupta

<https://github.com/sumanyugupta/CS1632_Deliverable3>

CS 1632 – Deliverable 3: Performance Testing

**Summary**

The most challenging part of this deliverable was figuring out how to parse the intricacies of each individual block in the blockchain protocol. Each line contained 5 elements that reflected varying kinds of information that proved difficult to accurately separate and then use to verify the blockchain. In addition, reducing the program’s runtime was always in the back of our mind so a lot of additional time on our end was spent whiteboarding and attempting to make methods simpler and use data structures that were not computationally expensive by themselves.

The good (and stressful) part about this project was that there were an extremely wide variety of edge cases and failure modes to consider. We focused on testing the input file a lot. For example, each block contained a timestamp value that relayed the time of transaction. One of our checks tested the value to make sure that only one period appeared in it. Since that period separated the seconds and nanoseconds value, too many or too few periods would indicate an invalid timestamp format and would need to be rejected accordingly. For transactions, we had to not only make sure that the overall transaction element was properly formatted, but we also had to identify if an address was too long, or if last transaction was not from SYSTEM, either of which would also invalidate the transactions. Finally, we also had to consider some really weird possibilities, such as the user not running the program in our directory set-up and rescuing that exception if it would ever occur.

Based on the flame graph, the method that took up the most amount of CPU time was our “compute\_hash” method. This method performed the complex math that involved using the first four elements of each block to calculate a hashed value using the algorithm provided. To be more specific, we believe that the two most CPU-intensive phases of the algorithm were raising each value in the string that is eventually hashed to the 2000th power and then computing a modulo with a large number such as 65536 to take the majority of time.

After calculating our mean runtime tested on the “long.txt” file and looking at the flame graph, we determined that that there were not any relatively faster methods to reduce the execution time of our “compute\_hash” method. In addition, a mean runtime of 25.908 seconds was relatively faster than other groups’ times so we decided to not change our program further.

**Flame Graph**

****

**Runtimes**

**Run 1:**

real 0m25.829s

user 0m25.063s

sys 0m0.219s

**Run 2:**

real 0m25.803s

user 0m25.109s

sys 0m0.172s

**Run 3:**

real 0m26.093s

user 0m25.313s

sys 0m0.234s

**Mean real time: (**25.829 + 25.803 + 26.093) / 3 = 25.908s

**Median real time:** 25.803s