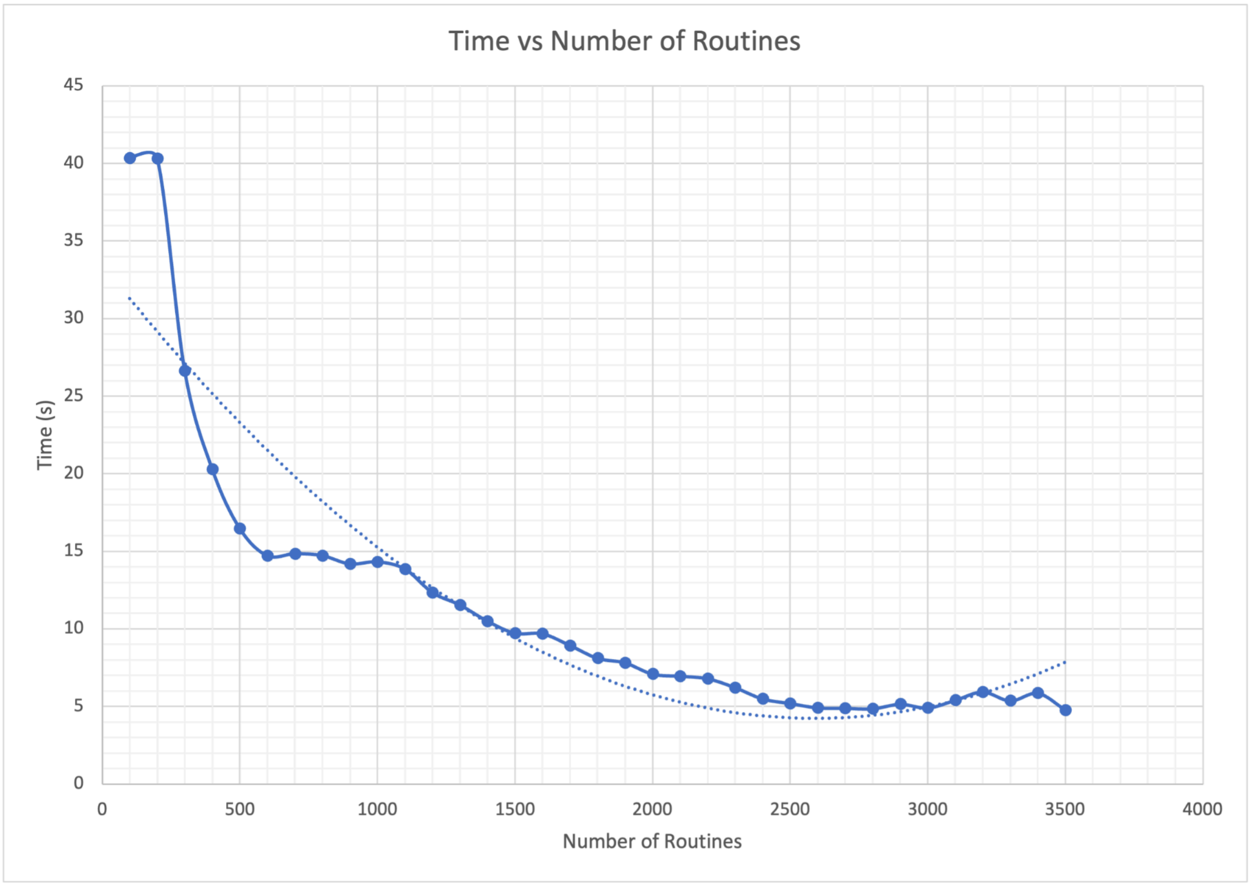
1. How do you determine the number of goroutines scanning hosts concurrently?

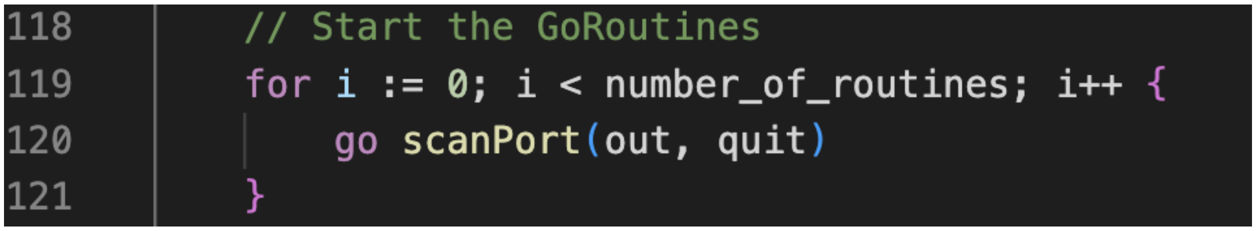
In line 32, we define the number of goroutines by the variable number_of_routines, which is dynamically set to 2500 at the beginning of the program. The value 2500 is chosen based on our experiment. Hence, regardless of the number of hosts or ports to scan, the program will always attempt to start 2500 goroutines to perform the scanning concurrently. If the resources exceed due to computer limitations, one can always decrease the number of routines. The time taken to complete the scan would decrease in a behavior similar to the one shown below (change the variable `number_of_routines`).



Benchmarked by scanning ports 0 to 500 of IP addresses of 1.1.1.1/24

2. How and under what conditions do you start the goroutines?

Goroutines are started in the parse_arguments function, specifically after the ports have been parsed and before the IPs are iterated over. This is done by the following loop:



```
    // Start the GoRoutines
    for i := 0; i < number_of_routines; i++ {
        go scanPort(out, quit)
    }
```

This loop starts number_of_routines (2500) goroutines of the scanPort function, each of which will listen on the out channel for host-port strings to process.

3. How do you determine if a goroutine has finished?

In this program, there's no explicit check to determine when an individual goroutine has finished its work because goroutines are designed to run indefinitely, listening on the out channel for new host-port strings to scan. They only terminate when the program itself exits. The program does not wait for goroutines to finish; instead, goroutines continuously listen on the out channel for work and process incoming tasks as long as the program is running. Hence, all the goroutines will have finished if the channel is empty.

4. Why is your solution deadlock-free?

A channel acts as a medium that enables the goroutine to act independently amongsdt each other while still maintaining no race conditions.

Because of this, the channel can help prevent deadlocks. The channel limits the number of goroutines that may access the channel at once, ensuring mutual exclusion amongst the goroutine.

5. Do you manage to run as many goroutines as possible at any time?

The program attempts to run a fixed number of goroutines (number_of_routines) concurrently, which is determined with regard to the workload (the number of hosts and ports to scan).

There are overhead associated with scheduling and switching between goroutines. As the number of goroutines increases, the scheduler has to do more work to manage these goroutines, which can lead to increased CPU usage and reduced efficiency due to context switching overhead. Hence, the number of goroutines is limited by system resources.