P3

edgardjfc

March 2022

1 Introduction

For this assignment on queuing theory we started off with a code by our teacher Dr. E. Shaeffer [1] . This base code allowed us to check for prime numbers in a list of numbers, organized in different ways, calculating the time it took to find the primes for each list. With the variable of having a different order.

In this assignment we had the objective to observe the statistical impact of different factors in the processing time of our code. In this case the variables were the function used to find the prime numbers, the organization of the list of numbers and the number of cores used for the process.

2 Code

```
prime <- function(n) {
    if (n < 4) {
        return(TRUE)
    }
    if (n %% 2 == 0) {
        return(FALSE)
    }
    for (i in seq(3, max(3, ceiling(sqrt(n))), 2)) {
        if (n %% i == 0) {
            return(FALSE)
        }
    }
    return(TRUE)
}
```

Here we can see the function A

```
primeAlt <- function(n) {
        if (n < 4) {
```

```
            return(TRUE)
        }
        for (i in 2:(n/2)) {
            if (n %% i == 0) {
                return(FALSE)
            }
        }
        return(TRUE)
}
```

And here we have the function B

For this part of the code we designed two different functions in order to find the prime numbers, with the objective to see which one gave us a better performance.

```
startingPoint <- 900
endPoint <-  1500
original <- startingPoint:endPoint
inverted <- endPoint:startingPoint
randomized <- sample(original)
repetitions <- 30

suppressMessages(library(doParallel))
registerDoParallel(makeCluster(detectCores()))
oaa <-  numeric()
iaa <-  numeric()
raa <-  numeric()
for (r in 1:repetitions) {
    oaa <- c(oaa, system.time(foreach(n = original,
                                      .combine=c) %dopar% prime(n))[3])
    iaa <- c(iaa, system.time(foreach(n = inverted,
                                      .combine=c) %dopar% prime(n))[3])
    raa <- c(raa, system.time(foreach(n = randomized,
                                      .combine=c) %dopar% prime(n))[3])
}
stopImplicitCluster()

summary(oaa)
summary(iaa)
summary(raa)
```

For this section of the code we designated the range of numbers in which we'll do a search for prime numbers, and then defined empty lists that will contain the time it takes for each iteration of the list. We repeated that part of the code several times in order to record every list of every permutation possible. Within these different permutations we had 2 different formulas to

find the prime numbers, 3 different orientations of the list of numbers (forward, backwards and randomized), and 3 different sets of cores used in the process (6, 3 and 1).

3 Results

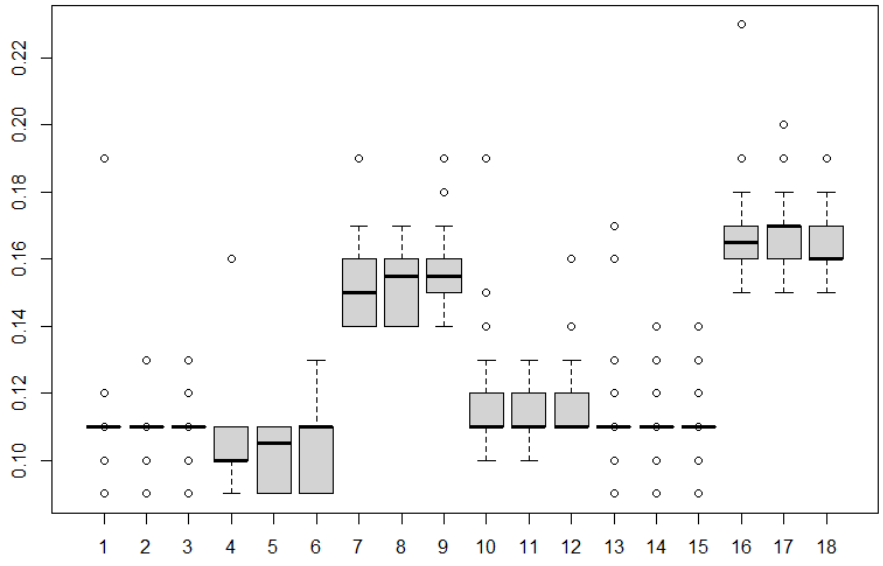

Figure 1: Plotted data

4 Interpretation

We can observe that in the data plotted in figure 1 we see that the boxes 1, 2 and 3 (function A and 6 cores of processing) and 10, 11 and 12 (function B and 6 cores of processing) have no significant difference, due to the processing time being too fast to give any significant result.

In the boxes 4, 5 and 6 (function A and 3 cores) and 13, 14 and 15 (function B and 3 cores) we see a difference in the processing time, where the function B is consistently slower than the function A.

We see the most pronounced results in the boxes 7, 8 and 9 (function A and 1 core) compared to 16, 17 and 19 (function B and 1 core), where the difference in the time it takes is a lot more significant, showing that the formula A is the more efficient of the 2.

We can also observe the clear improvement in processing time when using multiple cores to process this code

5 Conclusion

We can conclude that there is an important need on creating code that can be efficient and give us the best performance possible, since poorly written code can give us the results we want, at the cost of poor performance or possible bugs in the code.

References

[1] E. Schaeffer. *GitHub,Queuing Theory*. URL: `https://github.com/satuelisa/Simulation/tree/master/QueuingTheory`.