Assignment 4, Due December 2

1 Overview

This is the third MPI programming project. You are to write an MPI program to solve the ***parking garage problem***, detailed below. The parking garage problem is to determine the average number of parking spaces occupied by cars and the probability of a car's being turned away because the garage is full, given information about the garage and the characteristics of car arrival times.

2 Project Description

2.1 Problem Description

A garage has a fixed number of stalls S. Cars arrive at the garage at random intervals, with the inter-arrival times being exponentially distributed with mean α. When a car arrives at the garage, if a stall is available, it parks there. The number of minutes that it stays parked is a random variable, normally distributed with mean μ and standard deviation $\mu/4$. If there are no stalls available, the car leaves. We want to know the steady state values of two statistics related to this problem:

1. the mean number of stalls occupied by cars at any given time, and
2. the expected fraction of arriving cars that are unable to park because the garage is full.

These statistics depend upon the values of S, α,and μ. Your program's objective is to output an accurate estimate of these two statistics.

2.2 Usage

Name the executable `simulate_garage`. Proper usage is of the form

```
simulate_garage  <1|0> <num_stalls>  <exp_mean> <gauss_mean>
```

where

- if the first argument is `0`, the program's behavior is not repeatable, and if the first argument is `1`, it is repeatable.
- `num_stalls` is the number of parking spaces in the garage, henceforth called ***stalls***,
- `exp_mean` is the mean of an exponential distribution that governs the inter-arrival times, in minutes, between cars, and
- `gauss_mean` is the mean of a normal distribution with standard deviation `gauss_mean`/4 that governs the number of minutes that a car stays in the garage.

The program must detect incorrect command line usage and report it on standard output, including missing arguments, arguments that are non-numeric, incorrect value for the first argument, and a number of stalls that is not a positive integer.

The issue of repeatability is described below.

2.3 Output

If there are no usage errors, the output should be printed to standard output in the form:

```
expected occupancy:   x
expected fraction of cars turned away: y
```

where `x` and `y` are the estimates that your program computed, with 3 decimal digits of precision. It should also compute the total work performed by all processes, excluding the time to output the result, and ***print that to the standard error stream***:

```
total work:  w
```

where `w` is the total number of seconds used by all processes.

2.4 Parallelization Requirements and Considerations

You must design and implement a parallel program to obtain estimates of these statistics using the Monte Carlo methods we have covered in class. Only the root process is allowed to parse the command line, and only the root process is allowed to write to the output stream.

- The program must run a large enough number of iterations so that the change in both estimates is less than 10^{-4}. If for some reason, one or more estimates are not converging, it should stop after 10^8 iterations. You must decide how to best decompose this problem to take advantage of the parallel processes available to the program. They should be used to maximize the probability that the program reaches an accurate answer. Considering the fact that some sequences may not converge while others might, how should you take advantage of multiple processors?
- If the program is run with the repeatability argument equal to 1, then the random number generation should be repeatable from one run to another and the sequences of numbers that each process gets should be independent, uncorrelated, and repeatable as well. This means that the seed(s) are fixed for each run. If the repeatability argument is equal to 0, then then the random number generation should not be repeatable from one run to another. This means that the seeds can be chosen at run-time dynamically, such as by using the time or the process id assigned to the process, etc.
- If the program is run in repeatable mode, the program must produce the same results when run with any number of processes.
- I suggest running the program many times, collecting the outputs, and comparing outputs. Also, vary one parameter and see how the behavior changes as that parameter is changed.

2.5 Design Suggestions

Your program should model time in minutes with a real-valued variable t. Initially, $t = 0$. It should use an array `Garage[]` of size S to represent the time at which each garage stall is next available. Initially `Garage[k]` $= 0$ for all $k, 0 \leq k < S$.

The program must randomly generate inter-arrival times with mean α using the method of inversion. Suppose the current time is t_0. The program generates the next arrival time by generating a random inter-arrival time and adding it to t_0. Suppose that time is t_n. It then looks for a garage stall k such that `Garage[k]` $\leq t_n$. If it finds a free garage stall, say k, it updates `Garage[k]` by generating a normally distributed random variable with mean μ and standard deviation $\mu/4$ using the method of acceptance rejection, replacing it by the time when this car will leave the stall.

3 Program Grading Rubric

The program will be graded based on the following rubric out of 100 points:

- The program must compile and run on any `cslab` host. If it cannot be built on any `cslab` host, it loses 80 points.
- **Correctness** (60 points)
  - The program output should produce estimates of the statistics that are correct. Correctness will be determined against a "gold" version of the program run with sufficiently many iterations to converge.
  - The program should work correctly no matter how many processes it is given.
  - It should handle errors correctly.
- **Performance** (20 points)

  Has the program used the processes in a computationally beneficial way?
- **Design and clarity** (10 points)

  Is the program organized clearly? Are functions and variables designed in an appropriate way? If not, the program will lose points.
- **Compliance with the Programming Rules** (10 points)

  Are all of the rules stated in that document observed? Programs that violate them will lose points accordingly.

4 Submitting the Homework

This is due by the end of the day (i.e. 11:59PM, EST) on December 2, 2019. Follow these instructions precisely to submit it.

Your program should be a single source code file, written in C. It must have a `.c` suffix. It does not matter what base name you give it because the `submithwk` program will change its base name anyway to the form `username_hwk4.c` (or more accurately, `username_hwk4.`*XXX*, where *XXX* is the suffix you gave it.)

Assuming this file is in your current working directory and is named `myhwk4.c` you submit by entering the command

```
submithwk_cs49365 -t  4  myhwk4.c
```

The `-t` tells the command it is a plain text file. The program will copy your file into the `hwk4` sub-directory

```
/data/biocs/b/student.accounts/cs493.65/hwks/hwk4/
```

and if it is successful, it will display the message, "`File username_hwk4.c successfully submitted.`"

You can only run this command on a cslab host. You cannot run it on eniac, so remember to ssh into a cslab host before doing this!

You will not be able to read this file, nor will anyone else except for me. But you can double-check that the command succeeded by typing the command

```
ls -l /data/biocs/b/student.accounts/cs493.65/hwks/hwk4/
```

and making sure you see a non-empty file there.

If you put a solution there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date. You cannot resubmit the program after the due date.