Assignment 1

Running in the 90s

CS3210 – 2025/26 Semester 1

Learning Outcomes

This assignment aims to familiarize you with parallel programming and performance evaluation on a single node, multi-core system using OpenMP. We hope it reinforces your understanding of the process of parallelizing a sequential algorithm and gives a fun and satisfying introduction to parallel programming.

Changelog

12 Sep, v2: Changes in Section 2, specifically the pseudo-code. The car movements are done after all the acceleration and deceleration decisions of all cars are completed. Small typo fixes on the rules.
13 Sep, v3: Changes in Section 2, specifically the summary of the rules 2c and 3 to match the description.

This assignment is designed to assess your understanding on the usage of OpenMP.
Do **not** use POSIX threads, `fork()`, and/or C++ threads in your solution.

1 Problem Description

1.1 Introduction

With over 1 billion cars in use in the world, traffic management has been more important than ever in many places. With many factors affecting the flow of traffic, people often turn to traffic simulations, which abstracts away certain parts of traffic behaviour while retaining others which are central to traffic phenomenons observed.

In this assignment, you will implement and parallelize a modified version of the Nagel-Schreckenberg model for the simulation of freeway traffic. For clarification of doubt, you are to implement the Extended Augmented Nagel-Schreckenberg model, outlined in Section 1.4. Read through the entire assignment brief before attempting to come up with a solution.

The first part of the assignment brief addresses the problem to be solved; followed by an overview of the skeleton code we give to you. The last part of this assignment brief should address administrative matters for your submission.

1.2 Background: Nagel-Schreckenberg model

Nagel-Schreckenberg is a traffic simulation model which is used to generate traffic-like behaviour. While the model is simple, it is able to give interesting behaviour, in particular showing how traffic jams propagate.

This model is a 1-D (one-lane) model, with periodic boundary conditions, i.e. cars which has gone through the entire road will appear at the start of the road again – you can think of it as modelling a huge roundabout.

Suppose that each car c_i have position x_i and velocity v_i. Note that throughout this assignment, the position and velocity can be represented as non-negative integers. For a road of length l, we have $0 \leq x_i < l$.

This model consists of four steps:

1. (Acceleration) If v_i is less than the maximum velocity v_{max}, the car tries to speed up by 1 unit.
2. (Avoid crashing) If the distance to the car ahead, d, is less than or equal to v_i, then reduce v_i to $d-1$.
3. (Random deceleration) If $v_i > 0$, with some probability p_{dec}, v_i decreases by 1 unit.
4. (Movement) The car moves forward by v_i unit.

These four rules can be summarized as:

1. $v_i \leftarrow \max(v_{max}, v_i + 1)$
2. $v_i \leftarrow \min(d-1, v_i)$
3. $v_i \leftarrow \max(0, v-1)$ with probability p_{dec}
4. $x_i \leftarrow x_i + v_i$

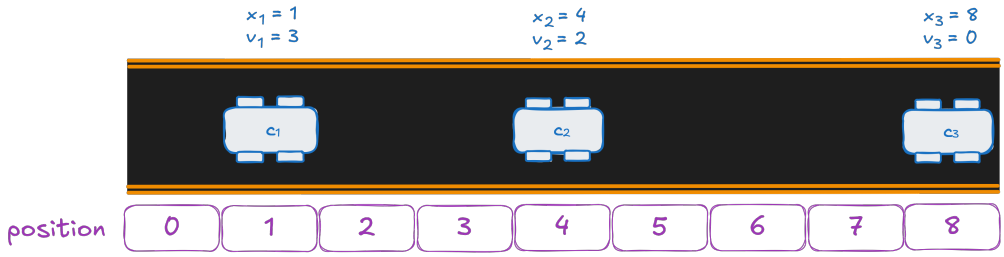
Note that these four rules ensures that there are no collisions nor overtaking of cars. To start this simulation, the following parameters need to be supplied:

- l, the length of the road; must be greater than 0
- n, the number of cars; must be less than l
- v_{max}, the maximum velocity of any cars; must be greater than 0
- p_{dec}, the probability of decelerating; must lie in $[0, 1]$
- t, the number of timesteps that the simulation is to be run for

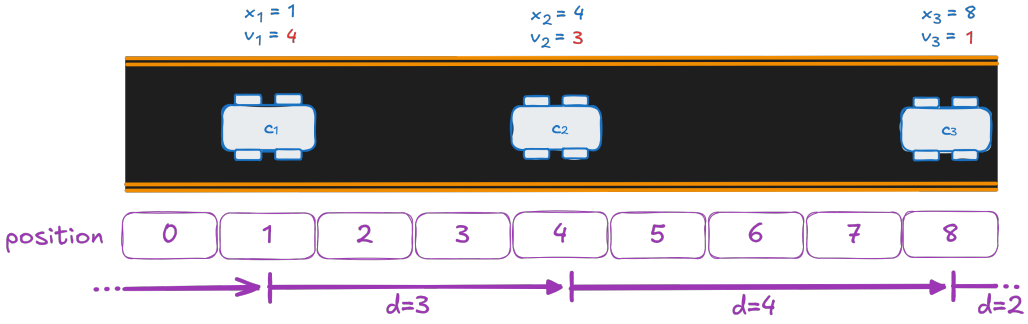
The simulation has two parts. The first part is the setup: given the parameters above, we generate the n cars with their initial positions and velocities. The second part is the actual simulation part: we apply rules 1-4 repeatedly for each of the t timesteps that the simulation is supposed to run for.

Below is an example simulation on what happens within one timestep, on the Nagel-Schreckenberg model. For the following illustration, $l = 9$, $n = 3$, and $v_{max} = 5$.

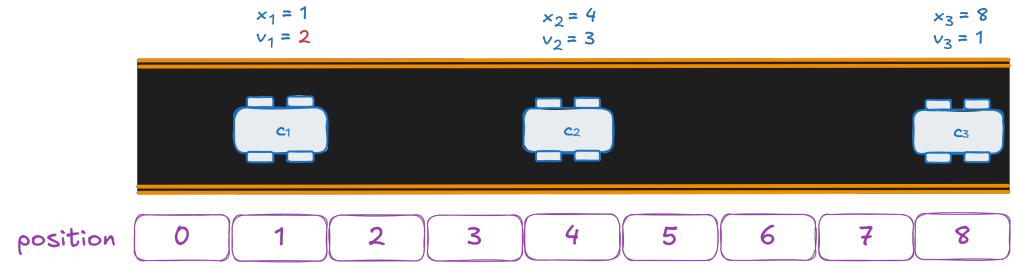
Start of timestep



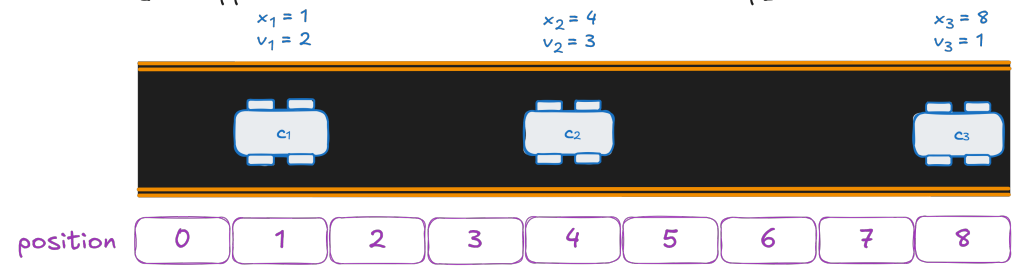
After applying rule 1



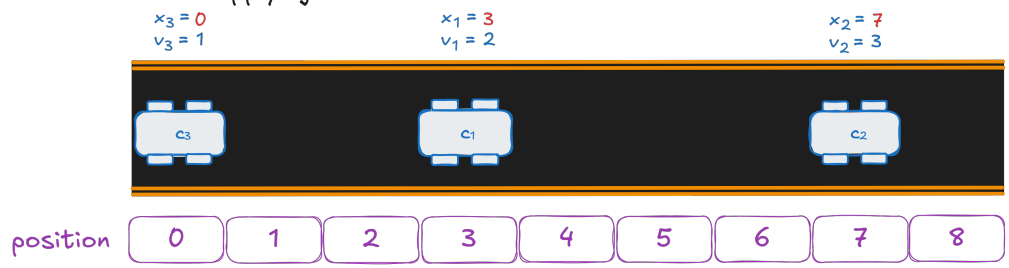
After applying rule 2



After applying rule 3
[It happens that no cars decelerate on this step]



After applying rule 4



1.3 Augmented Nagel-Schreckenberg model

In the previous subsection, you have seen the Nagel-Schreckenberg (NS) model, which would serve as our basis for the simulation in our assignment. While the NS model is able to exhibit certain traffic phenomenons, it is oftentimes too simple to be used as-is. Luckily, the simplicity of the model allows us to augment it with extra rules to better mimic real traffic.

We will have two modifications to the NS model: this section, which we refer to as the *Augmented Nagel-Schreckenberg Model*, is the first modification, built on top of the base NS model. The second modification outlined in the next section, which we refer to as the *Extended Augmented Nagel-Schreckenberg Model*, is built on top of the Augmented NS model.

Read through the entire assignment brief before implementing your solution. This assignment requires you to implement the *Extended Augmented Nagel-Schreckenberg Model* (next subsection), not the *Augmented Nagel-Schreckenberg Model* (this subsection).

1.3.1 Simulation Scenario and Rules

In the *Augmented Nagel-Schreckenberg Model*, the scenario remains the same – that is, a one-lane traffic, on a circular road, without any collision and overtaking. However, the rules that are carried out by each car is now augmented with further rules, one which affects how a car starts after it has been stationary; and another which affects how cars slow down. The following rules apply:

1. (Slow start) If v_i is 0 and $d > 1$
   - with probability p_{start}, the car accelerates normally; skip to step 3.
   - otherwise, v_i remains at 0 on this timestep, and will increase to 1 on the *next timestep*; skip to step 4.
2. (Avoid crashing) Let d be the distance between your car and the car in front of it; and let v_{i+1} be the velocity of the car in front of you.
   - If $d \leq v_i$ and either $v_i < v_{i+1}$ or $v_i < 2$ (i.e. the car in front is moving faster than you, or the car in front is very close to you), then decrease v_i to $d - 1$.
   - If $d \leq v_i$, $v_i \geq v_{i+1}$, and $v_i \geq 2$ (i.e. the car ahead is slower and your speed is substantial), then decrease v_i to $\min(d - 1, v_i - 2)$
   - If $v_i < d \leq 2v_i$ and $v_i \geq v_{i+1}$ (i.e. the car ahead is moving slower than you, and you forsee a collision in 2 timesteps), then set v_i to $v_i - \left\lfloor \left(\frac{v_i - v_{i+1}}{2}\right) \right\rfloor$
3. (Acceleration) If v_i has not been modified by rules 1 and 2, $v_i < v_{max}$, and $v_i + 1 < d$, set v_i to $v_i + 1$.
4. (Random deceleration) If $v_i > 0$, v_i decreases by 1 unit with some probability p_{dec}.
5. (Movement) Move the car forwards by v_i unit.

To summarize:

1. If $v_i = 0$ and $d > 1$
   (a) skip to step 3 with probability p_{start}
   (b) otherwise, $v_i \leftarrow 0$ this timestep, and $v_i \leftarrow 1$ in the next timestep and skip step 3
2. There are 3 cases of deceleration due to the next car:
   (a) If $d \leq v_i$ and either $v_i < v_{i+1}$ or $v_i < 2$; then $v_i \leftarrow d - 1$
   (b) If $d \leq v_i$, $v_i \geq v_{i+1}$, and $v_i \geq 2$; then $v_i \leftarrow \min(d-1, v_i - 2)$
   (c) If $v_i < d \leq 2v_i$ and $v_i \geq v_{i+1}$; then $v_i \leftarrow v_i - \left\lfloor \left(\frac{v_i - v_{i+1}}{2}\right) \right\rfloor$
3. If v_i has not been modified by rules 1 and 2, then $v_i \leftarrow \min\left(d-1, \min(v_i + 1, v_{max})\right)$
4. $v_i \leftarrow \max(0, v_i - 1)$ with probability p_{dec}
5. $x_i \leftarrow x_i + v_i$

1.3.2 Usage of PRNG

In steps 1 and 4, the acceleration / deceleration is not deterministic, i.e. the cars accelerate / decelerate with some probability. In this case, psuedo-random number generators (PRNGs) with seeds are used in order to introduce the randomness. The PRNG is used in the following manner: suppose we want to see whether the random deceleration happens (step 4) with probability p_{dec}.

1. We ask the PRNG to generate a number between 0 and 1 uniformly
2. If the number is less than p_{dec}, then we decelerate

To simplify this process, we provide the utility function `flip_coin(p_dec, traffic_prng::engine)`, where `traffic_prng::engine` is the PRNG engine.

Note that for each timestep, each car would need at most 2 calls to the PRNG (one in step 1, and one in step 4). For ease of implementation, your implementation should call the PRNG *exactly twice* for each car, once for slow start and once for deceleration, in order of the cars (i.e. the first car will call the PRNG twice, then the second car will call the PRNG twice, ...). The first result will be used for step 1 (if applicable), and the second result will be used for step 4 (if applicable).

In this assignment, we employ a PRNG with a *seek* function, which allows you to find the n^{th} result of the PRNG in $O(\log n)$ time. This can be done by calling `traffic_prng::engine.discard(n)`. This information might be helpful when you are trying to come up with a parallel implementation which behaves exactly as the reference program given.

1.3.3 Input parameters

With the augmented NS model, we have one extra parameters than the base model. The parameters that we use in this version are as follows:

1. l, the length of the road; must be greater than 0
2. n, the number of cars; must be less than l
3. v_{max}, the maximum velocity of any cars; $0 < v_{max} < l$
4. p_{dec}, the probability of decelerating; must lie in $[0, 1]$

5. p_{start}, the probability that the car will start quickly; must lie in $[0, 1]$
6. t, the number of timesteps that the simulation is to be run for
7. s, the seed of the PRNG

1.3.4 Initial Car Configuration

You will be supplied with a `std::vector` of `Cars`, which represents the initial configurations of the cars. Each `Car` is a `struct` containing 4 pieces of information:

1. `id`, which is the ID of the car. This ranges from 0 to $n - 1$.
2. `v`, which is the velocity of the car.
3. `position`, which is the position of the car. There will be no overlapping cars, i.e. this value would be unique for all cars. This value ranges from 0 to $l - 1$.
4. `lane`, which is the lane of the car. For the single-lane traffic scenario, this value would only be `0`.

1.4 Extended Augmented Nagel-Schreckenberg model

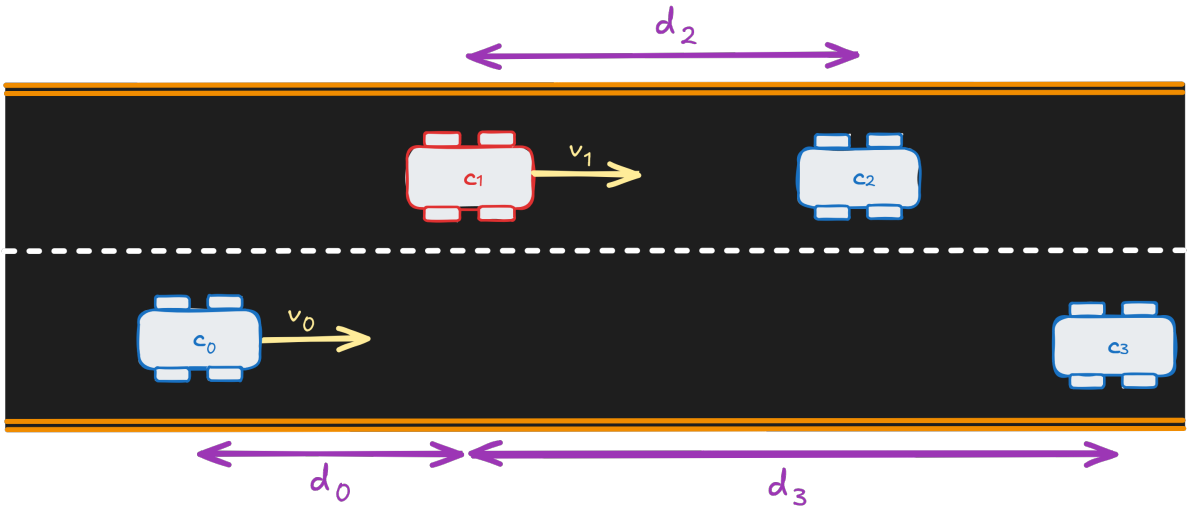
This subsection outlines the rules for *Extended Augmented Nagel-Schreckenberg model*, which is built on top of the *Augmented Nagel-Schreckenberg model*. All the modifications done in the previous section still holds; in this second modification, we further modify *scenario* of the simulation.

1.4.1 Updated simulation scenario

In the *Extended Augmented Nagel-Schreckenberg model*, the traffic simulation model now has **two-lane, one-way traffic**, where the cars can now change lane and overtake other cars.

1.4.2 Lane-change rules

We will use the following illustration for ease of explanation of the overtaking rules:



We focus on the lane-changing rules on car c_1. In the diagram above, car c_2 is the car immediately ahead on the same lane as c_1; car c_3 is the car immediately ahead of c_1 if it changes lane; and car c_0 is the car immediately behind c_1 if it changes lane. Car c_1 will change lane if **all** of the following are fulfilled:

1. $d_2 < d_3$; i.e. there's more space in the other lane than in the current lane

2. $v_1 \geq d_2$; i.e. the car needs to decelerate if he doesn't change lane
3. There is empty space at the car's position in the other lane
4. $d_0 > v_0$; i.e. the car avoids getting rear-ended

For ease of implementation, all cars would check whether they are going to change lanes simultaneously based on the final locations and velocities of the cars from the previous timestep.

The decision whether to change the lane or stay on the same lane occurs before the resolution of the velocities and positions (rules 1-5 in Section 1.3.1)

In summary, for each timestep:

1. All cars would check whether they would want to change lanes
2. Any cars which wants to change lane would change its lane
3. All cars would set its new velocity (rules 1-4 in 1.3.1)
4. All cars would move forwards based on its velocity (rule 5 in 1.3.1)

1.4.3 Initial Car Configuration

The initial car configuration would remain the same, except for the presence of two lanes. In the EANS model, the lane number can now be either 0 or 1.

2 Implementation Guidelines

You are required to make a parallel implementation of the *Extended Augmented Nagel-Schreckenberg model* (EANS model) which behaves exactly as the reference executable(s).

The reference executables have implemented the EANS model as outlined in the previous section. Aside from the general implementation, there are some implementation details which are mentioned; to ensure that your implementation behaves in the exact same manner as the reference executables, do take note of these particular implementation details (reproduced from Section 1):

1. For each timestep, each car calls the PRNG twice, regardless of whether the car needs to check for the slow-start or the random deceleration rule
2. All lane change decisions are done at the same time; i.e. each car checks for the lane-change criteria based on its placement in the end of previous timestep; this is then proceeded by all of the lane changes that needs to be made.

To help with implementation, we give the general algorithm for a sequential reference executable in the next page. You are free to implement any other algorithms, use supplementary data structures, and employ other strategies, as long as the output for your parallel program is the same as the executables given.

The following is the algorithm for one of the sequential reference executables:

input: n, number of cars
input: t, total number of timesteps
input: other parameters
input: `std::vector<Car>` cars c_0 to c_{n-1}

while *timestep* $< t$ **do**
  for $i \leftarrow 0$ **to** $n-1$ **do**
    car c_i calls the PRNG twice;
    $ss_i \leftarrow$ result of first PRNG call;
    $dec_i \leftarrow$ result of second PRNG call;
  end
  for $i \leftarrow 0$ **to** $n-1$ **do**
    car c_i checks whether it wants to change lane;
  end
  for $i \leftarrow 0$ **to** $n-1$ **do**
    car c_i moves lane, if it wants to;
  end
  for $i \leftarrow 0$ **to** $n-1$ **do**
    `// rule 1: slow start`
    if c_i *is not moving and can move forward* **then**
      determine whether car c_i starts slow using ss_i;
      skips to the necessary section;
    end
    `// rule 2: deceleration`
    if c_i *fulfills a deceleration criteria* **then**
      c_i decelerates accordingly;
    end
    `// rule 3: acceleration`
    if v_{c_i} *should accelerate* **then**
      c_i accelerates;
    end
    `// rule 4: random deceleration`
    if c_i *can decelerate* **then**
      use dec_i to determine whether c_i decelerates;
    end
  end
  for $i \leftarrow 0$ **to** $n-1$ **do**
    `// rule 5: movement`
    car c_i moves forward;
  end
end

3 Grading

You are advised to work in groups of two students for this assignment (but you are allowed to work independently as well). You are not required to have the same teammate for the following assignments. You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalised. If you use external references, cite them or at least mention them in your report (what you referenced, where it came from, how much you referenced, etc.). *This also includes use of AI tools such as ChatGPT, Copilot, etc.* Please refer to our policy in the introductory lecture on AI tool usage.

The grades are divided as follows:

- 7 marks – your parallel implementations with OpenMP, split into:
  - 3 marks – the *correctness* of your simulation and implementation, described in Section 3.1.
  - 4 marks – the *performance* of your simulation, described in Section 3.2.
- 5 marks – your report, described in Section 3.3.
- Up to 2 bonus marks, described in Section 3.5

3.1 Correctness Requirements

In this section, we define the correctness requirements for your program. These requirements are in *addition* to any other information presented earlier in this document.

Your implementations should:

- Be written in C or C++, and compile and run successfully with `gcc/g++` **and** `clang/clang++` with flags `-std=c++20 -fopenmp -O3` on all PDC lab machines.
- Create a simulation executable named `sim` when you run `make` in the top level of your submission.
- Be parallelized using OpenMP *only* – you are not allowed to use any other parallelization libraries, frameworks, or even C++ threading libraries or synchronization like `std::mutex`. You are also not allowed to submit a fully-sequential simulation – your program must show speedup over a single-threaded execution of your program.
- Be tested and run via Slurm – we will use the default CPU mapping and bindings for `srun`.
- Have no memory leaks or undefined behavior.
- Not use any external utilities or libraries (e.g., those not provided with the C/C++ environment) that are not otherwise approved by the teaching team. Such approvals will be listed publicly in the FAQ document mentioned later. If what you want is not listed, you may request permission via our contact details in Section 4.2. Note that we are unlikely to allow things that directly impact performance.
- Use the provided `common.h` and `common.cc` files *without any modifications*, as we don't want you to focus on this. You may change other parts of our code skeleton assuming that all other requirements are fulfilled. Our utility scripts can all be modified freely.
- **Have the same output compared to the one generated by our implementation**
  - We will check your code on test cases provided to you, and a number of private test cases.
  - We may update the our implementation if we discover any issues with it w.r.t our requirements.
- Not have any race conditions, data races, deadlocks, or any other synchronization issues.

3.2 Performance Requirements

We provide a number of **benchmark executables** to compare your solution against. Each are named in the format `bench_i`, where i is an integer. The i value does not necessarily represent any specific ordering of the benchmarks. You will be assigned performance marks based on your performance against these reference benchmarks. We do not disclose the specific mark allocation per benchmark.

Performance Metric: We will compare the wall-clock runtime of your program against the wall-clock time of each of the benchmarks. For each run, we will run each program with the same input file and the same number of threads. We will run each program a few times (unspecified) and take the average (mean) time as the final time.
Requirements: To receive marks for a specific benchmark, you must be faster than it under these conditions:

- Your program will run with n threads, where n is the number of hardware threads on a machine.
- Your program will be run via Slurm.
- We reserve the right to deduct performance marks if there are correctness issues with your code.
- 80% of your performance grade will be from these specific executables, testcases, and machine types:
  - `bench-5.perf`; `inputs/in_1e7`; xs-4114 and w5-3423
  - `bench-4.perf`; `inputs/in_5e6`; i7-13700
  - `bench-3.perf`; `inputs/in_5e6`; w5-3423 and i7-7700
  - `bench-2.perf`; `inputs/in_1e6`; i7-13700
  - `bench-1.perf`; `inputs/in_1e4`; i7-7700 and xs-4114

  Note that the scores are not distributed uniformly across these combinations, i.e. some configurations will have greater weight than others.
- 20% of your performance grade will come from our hidden testcases. The input will have these guarantees to ensure a high-enough workload, and therefore can benefit from parallelism:
  - $n > 5000$
  - $L > n \times 10$
  - $p_{dec} \leq 0.9$
  - $p_{acc} \leq 0.9$
  - $t \geq 100$

We provide some reference test cases in the `tests/` directory. You can run the benchmarks with the provided `run_bench.sh`. **Note that the provided tests are not exhaustive, and therefore you should test your program with a variety of inputs to ensure correctness and performance.**. To generate new testcases, we have provided you with a testcase generator called `gen.py`.

We have tested our benchmark implementations, but as always, they are possibly incorrect. If you notice any issues, please do let us know. Our contact details are available in Section 4.2.

Slurm Usage
We reiterate that you should test and run your programs via Slurm. We will run your programs via Slurm, and if your program does not run correctly via Slurm, you will not receive marks. Furthermore, the CS3210 cluster will be heavily utilized during the assignment period, so Slurm allows a fair allocation of testing resources. **Do not test your program on login nodes!**

"Spirit of the Assignment" Note

We included a number of rules and constraints in this assignment for clarity. However, this is not a legal document – we cannot and do not want to cover every single eventuality in writing, and we also have certain learning objectives for you, so bypassing the spirit of the rules is not what we're aiming for.

At the same time, we want to encourage exploration and interesting solutions. If you believe you have an interpretation of any these rules that is different from the one we might have intended, **please ask us for clarification** (even privately if necessary). For avoidance of doubt, if you have any concern that you are not interpreting the requirements as intended, you must ask us.

3.3 Report Requirements

3.3.1 Format

Your report should follow these specifications:

- Five pages maximum for main content (excluding appendix).
- All text in your report should be minimum 11-point Arial (any typeface and size is ok so long as it's readable and not trying to bypass the page limit).
- All page margins (top, bottom, left, right) should be at least 1 inch (2.54cm).
- Have visually distinct headers for each content item in Section 3.3.2.
- It should be self-contained. If you write part of your report somewhere else and reference that in your submitted "report", we reserve the right to ignore any content outside the submitted document. An exception is referencing a document containing measurement data that you created as part of the assignment - we encourage you to do this.
- If headers, spacing or diagrams cause your report to *slightly* exceed the page limit, that's ok - we prefer well-organised, easily readable reports.

3.3.2 Content

Your report should contain:

- (1 mark) A brief description of your implementation, including:
  - An overview of your chosen algorithm, data structures, and parallelization strategy, and why these were chosen.
  - What OpenMP constructs did you use, and why did you use those specific constructs.
  - How work is divided among threads.
  - How you handled synchronization in your program.
  - How and why your program's performance scales with the number of threads. Vary the number of threads and present the data and trends clearly.

  Include any relevant details you think will help us understand your implementation.
- (2 marks) Description, visualization, and data on your execution, including:
  - How and why your program's performance changes based on parameters in the input file.
  - How and why your program's performance is affected by the type of machine you run it on. Include a comparison of at least *three* different hardware types.

- (2 marks) Describe at least TWO performance optimizations you tried, including supporting measurements and hypotheses on why they worked or did not work.

Please make sure to support your statements with clear data – tables, graphs, etc. We are looking for good quality, scientifically sound reports.

Additionally, your report should have an appendix (does not count towards page limit) containing:

- Details on exactly how to reproduce your results, e.g. nodes, inputs, execution time measurement, etc.
- Relevant performance measurements, if you don't want to link to an external document.

Tips:

- There could be many variables that contribute to performance, and studying every combination could be highly impractical and time-consuming. **You will be graded more on the quality of your investigations, not so much on the quantity of things tried** or even whether your hypothesis turned out to be correct.
- Performance analysis may take longer than expected and/or run into unexpected obstacles (like your program failing halfway). **Start early** and test selectively.

3.4 Skeleton Code

We provide skeleton code that can be used as a starting point for your implementation. The code is written using C++, but feel free to change this code, including changing to C-style C++ (though other languages are not acceptable), as long as all other requirements are met. Furthermore, you might consider writing your program using a mix of C and C++.

The skeleton code provided includes the files listed in Table 1:

File name	Description
`common.cc` `common.h`	(Do not modify) Common library; contains definitions and implementations for I/O and the PRNG used.
`simulation.cc`	These files constitute the main logic of the simulation, and where we expect most or all of your modifications to take place.
`executables/`	Executables for correctness checks and for comparison against the speed of your code.
`Makefile`	Default Makefile for compiling the code. Modify this as you make more source and header files.
`gen.py`	Script to hep you generate testcases.
`input.zip`	Example testcases.
`outputs`	Correct outputs for our example testcases (debug only).
`run_bench.sh`	Example script to run your program on Slurm against the existing benchmarks, on a specific machine type.

Table 1: List of provided files in the skeleton

3.4.1 Quickstart

1. Run `make` in the folder with the `Makefile`
2. This will produce two executables from your code: `sim.debug` and `sim.perf`. The `sim.debug` outputs the states of the cars at each step and is used for debugging and correctness checks, while `sim.perf` will have those calls removed automatically and is used for performance checks.
3. Check the correctness of your code by running `./sim.debug inputs/in_1e4` for example, and compare the `stderr` output with the given sequential implementation. To do so, run `./executables/sequential.debug` on the same input, then run `diff` on the two outputs. We have given you some sample outputs in the folder `outputs/` for your convenience.
4. Run a performance benchmark by running `./run_bench.sh i7-7700 inputs/in_1e4`. You'll notice that your code might "beat" some of the benchmarks!
5. Go forth and make your code correct, and then fast!

3.4.2 Using the `run_bench.sh` benchmark runner

The `run_bench.sh` script is used to run your program against the provided benchmarks. The script has the following usage: `./run_bench.sh <machine> <input_file>`. The script will run your program with the given input file on the specified machine type via `srun`. It will compare the runtime (but not the correctness) of your program against the provided benchmarks by running each three times. We may not use this exact script in our grading, and it's more for your convenience.

Caching results (important!): Our script caches the results of benchmark runs in a subfolder called `checker_cache/`. This is to avoid running the benchmark multiple times for the same input and machine type (since the results should not substantially change, as our benchmarks don't change). If you would like to clear this cache for any reason, *delete the checker_cache folder*.

3.4.3 Using and Generating Test Cases

We provide some starter test cases in the `inputs/` subfolder. We also provide a Python script (`gen.py`) to help you generate more test cases. You can run it as such:

```
python3 gen.py --n <number of cars> --L <Length of road> --vmax <V_max> --p-dec <p_dec>
--p-start <p_start> --steps <epoch count> --pos {random | even} --vel {random | zero} --seed
123123 --out <output file>
```

A testcase with these parameters will be printed to `<output file>`, with car velocities and positions either randomized or deterministic, depending on your given argument. A usage example would be:

```
python3 gen.py --n 100 --L 200 --vmax 10 --p-dec 0.5 --p-start 0.6 --steps 5 --pos random
--vel random --seed 32103210 --out inputs/my_input
```

This will output a testcase to `inputs/my_input` with 100 cars with $L = 200$, $v_{max} = 10$, $p_{dec} = 0.5$, etc. The velocities and positions of the cars will be generated randomly, based on the seed. The same seed will be used as an argument to your simulator.

3.5 Bonus - Speed Contest

You may obtain up to 2 bonus marks for achieving some of the fastest implementations in the class. More details can be found in the Assignment 1 Bonus section of the Student Guide (may take some time to be updated)

4 Admin

4.1 Accessing the Skeleton Code

We will use GitHub Classroom for this assignment. **Name your team** `a1-e0123456` (if you work by yourself) or `a1-e0123456-e0654321` (if you work with another student) – substitute your NUSNET number accordingly. You can access the Classroom via https://classroom.github.com/a/qX-krvpn. We will provide the skeleton code for the assignment through this repository.

You **must** use only the GitHub Classroom repository during this assignment. That is, you should not create other repositories for your codebase.

4.2 FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered in this document here. The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions**.

If there are any questions regarding the assignment, please use the Discussion Section on Canvas (preferred, as we need to disseminate information to students regardless) or email Theodore (mailto:theo@nus.edu.sg).

4.3 Deadline and Submission

Assignment submission is due on **Wednesday, 24 September, 2pm** (note: **not midnight!**).

You are required to do **two important steps** for submission.

- **GitHub Classroom**: The implementation and report must be submitted through your GitHub Classroom repository.
  - Push your **code and report** to your team's GitHub Classroom repository.
  - Your report must be a PDF named `<teamname>.pdf`. For example, `a1-e0123456-e0654321.pdf`. `<teamname>` should **exactly match** your team's name; if you are working in a pair, please DO NOT flip the order of your NUSNET IDs.
  - **Tag the commit** that you want us to grade with `a1-submission`; if you forget to add such a tag, we will be forced to use the most recent commit.
- **Canvas Quiz for Assignment 1**: Take the Assignment 1 quiz on Canvas and provide *both* the **name of your GitHub Classroom repository** and the **commit hash** corresponding to the `a1-submission` tag; if you are working in a team, only one team member needs to submit the quiz. If both of you submit, we will take the latest submission.

A penalty of 5% per day (out of your grade for this assignment) will be applied for late submissions.

Final check: Ensure that you have submitted to both **Canvas** and **GitHub Classroom**. Canvas should contain your commit hash and repository name, and GitHub Classroom should contain your code and report.