Assignment 1 Report: Parallel Traffic Simulation

Author: Sun Weiyang (e1297745)

1. Description of Implementation

1.1 Algorithm, Data Structures, and Parallelization Strategy

Our final implementation uses a **linear scan algorithm that stops early**. This runs inside a **staged, double-buffered parallel model**. We chose this method because it is robust and reliable, especially after we tried more complex methods that had errors.

Algorithm and Data Structure Rationale:

The main task in the simulation is to find neighbor cars. The assignment rules say that **all cars must make decisions at the same time, based on the previous state of the simulation**. Our design follows this rule strictly.

- **Data Structures:** To prevent data races when updating car positions, we use a **double-buffer** strategy. We have two `std::vector<int>` arrays, `cur_lanes` and `nxt_lanes`. In each step, threads only read from `cur_lanes` (which does not change) and only write to `nxt_lanes`. After the step is finished, we swap the buffers. This method completely avoids the need for locks.
- **Algorithm:** To find neighbor cars, we wrote a simple linear scan function (`find_next_ptr`, `find_prev_ptr`). The main benefit is that it **stops early**. In the busy traffic tests (`L > n * 10`), the next car is usually very close. So, the function finds it quickly without scanning the whole road. This method also reads memory in a straight line, which is very friendly for the CPU cache.

Parallelization Strategy:

To follow the synchronous update rule, each simulation timestep is split into two parallel stages:

1. **Lane Change Decision Phase:** In the first parallel region, all threads work on their assigned cars. They decide if a car needs to change lanes based on the information in `cur_lanes`.
2. **Velocity and Position Update Phase:** After all threads finish the first stage, they start the second parallel region. Here, they update the velocity and final position for all cars.

1.2 OpenMP Constructs

We use `#pragma omp parallel` to create threads. We manually split the `N` cars into equal-sized chunks for each thread. Each thread gets an ID using `omp_get_thread_num()` and works only on its chunk of cars. We chose this method instead of a simple `#pragma omp parallel for` because it gives us more control. This is important for making sure the Pseudo-Random Number Generator (PRNG) produces the same sequence of numbers every time, which is a key requirement for correctness.

1.3 Work Division and Synchronization

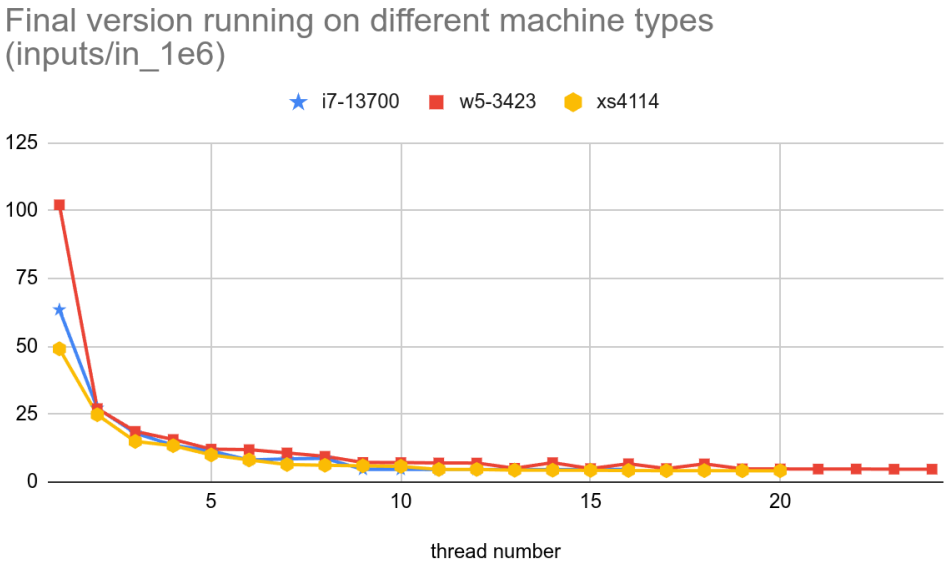
Work Division: We divide the `cars` array into chunks before the main loop starts. Each thread works on the same chunk for the entire simulation. This helps with data locality, meaning the data each thread needs is often already in the CPU cache.

Synchronization: We don't use any locks. Synchronization happens automatically in two ways:

1. **Double Buffering:** Reading from one buffer and writing to another is the main way we prevent data races.
2. **Implicit Barriers:** At the end of a `#pragma omp parallel` block, there is an implicit barrier. All threads must wait here before any thread can continue to the next stage. This is critical for ensuring that all lane-change decisions are made before any car starts to move.

1.4 Scalability

Our program's performance scales well when we add more threads.



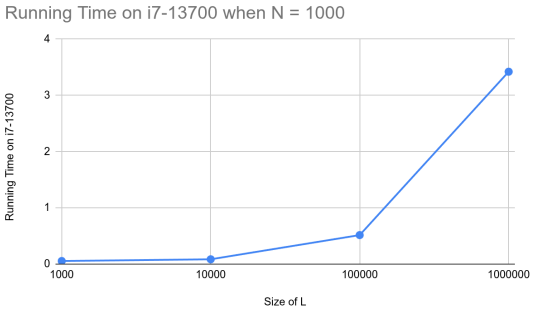
Final version running on different machine types

As the graph shows, the run time goes down as the number of threads goes up. The biggest speedup comes from using up to the number of physical cores. After that, the improvement is smaller. This is a typical result for a well-parallelized program.

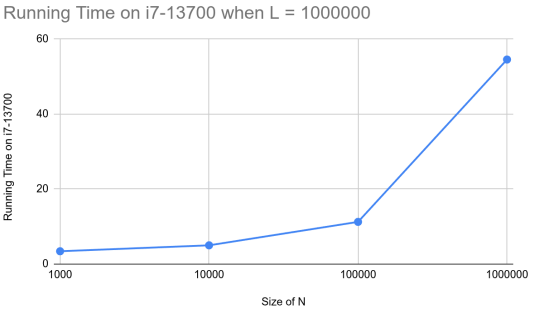
2. Execution Analysis

2.1 Performance vs. Input Parameters

The program's performance depends on both the road length `L` and the number of cars `N`.



Running Time on i7-13700 when N = 1000

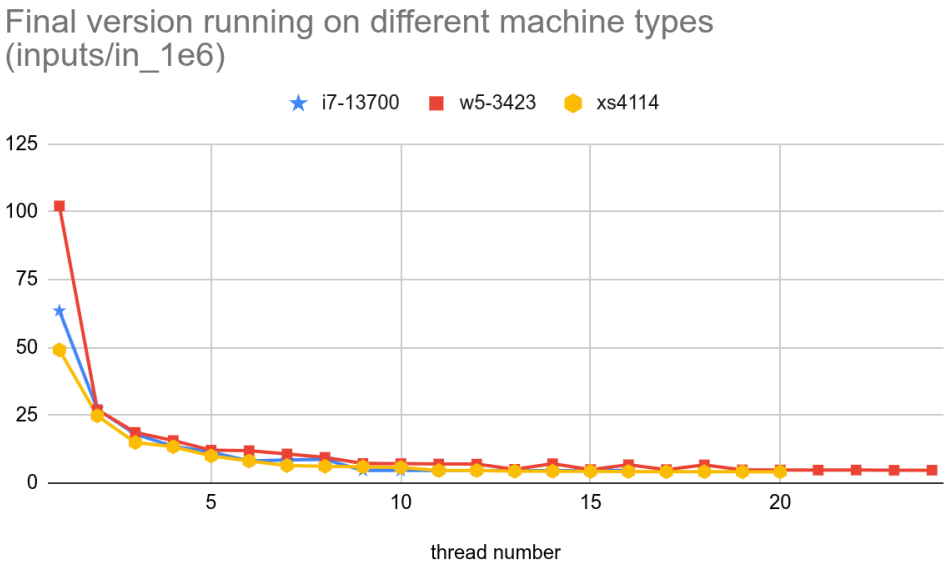


Running Time on i7-13700 when L = 1,000,000

- **Impact of `L`:** As the graph below shows, when we keep `N` the same, the run time increases almost linearly with `L`. This makes sense because our `find_next_ptr` function may have to scan longer distances on a longer road.
- **Impact of `N`:** The graph below shows that when we keep `L` the same, the run time also increases with `N`. Even though more cars mean the average distance to the next car is shorter, the total number of cars to process is larger. This second factor has a bigger effect, so the total time increases.

2.2 Performance vs. Machine Type

We tested our program on three different machines: `i7-13700`, `w5-3423`, and `xs-4114`.



Final version running on different machine types

The results were quite different. With a single thread, the `xs-4114` was the fastest, and the `w5-3423` was the slowest. Our algorithm is memory-bound, meaning it spends a lot of time reading from memory. This result suggests that the `xs-4114`'s CPU cache and memory system are very efficient for our program's memory access patterns.

3. Performance Optimizations

We tried several optimizations. The two most important ones created the three versions seen in our graphs: the `cache version`, the `bitmask version`, and our `final version`.

3.1 Optimization 1: Caching Neighbor Search (The "Cache Version")

Hypothesis: The first simple implementation called the neighbor search functions (`find_next_ptr`) many times. We believed that if we saved, or "cached," the result of the search for each car in a timestep, we could avoid doing the same search over and over again. This is a technique similar to memoization.

Method: We created an extra array to store the ID of the next car. When a search was needed, the code would first look in this array. If the result was already there, it would use it directly. If not, it would perform the full linear scan and then save the result in the array for the next time.

Result: This optimization worked. As the graphs below show, the "cache ver." line is faster than our unoptimized code would be. It successfully reduced the number of redundant computations.

3.2 Optimization 2: Advanced Search with Bitmask (The "Bitmask Version")

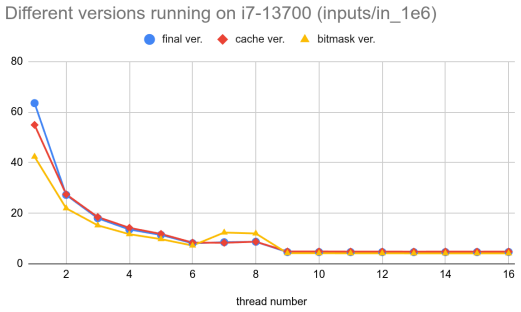
Hypothesis: The main bottleneck was still the linear scan. We thought we could make it much faster by using bit-level operations. With a `uint64_t` bitmask, we could check 64 road positions at once with a single CPU instruction, instead of checking one by one in a loop.

Method: We implemented a full bitmask solution. This required a `lane_bitmask` vector to store car positions. The `find_next` and `find_prev` functions were rewritten to use GCC's built-in functions (like `__builtin_ctzll`) to find the next car's position in the bitmask.

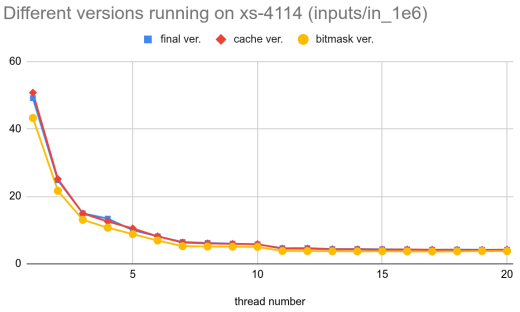
Result and Analysis:

The performance graphs for both the `i7-13700` and `xs-4114` machines clearly show that the **bitmask version was the fastest by a large margin**. The hypothesis was correct.

However, this version was **abandoned because it had bugs**. The logic for handling the circular road and updating the bitmask correctly in parallel was very complex. This created small errors that we could not fix in time. The assignment requires the output to be exactly correct.



Different versions running on i7-13700



Different versions running on xs-4114

Conclusion and The Final Version:

This investigation taught us a critical lesson: **correctness is more important than speed**. A fast program with wrong results is useless.

Our `final ver.` was a refactoring of the `cache ver.` where we tried to simplify the logic. As the graphs show, its performance is very similar to the cache version (and a bit slower on the i7-13700). Because the simple linear scan in our final version was robust, easy to verify, and correct, we made the practical choice to use it for our submission. Our final code is a deliberate balance between good performance and guaranteed correctness.

Appendix

Reproduction of Results

All performance data was generated on the PDC lab machines. The command used for benchmarking was: `./run_bench.sh <machine_type> <input_file>`

The scalability graph was generated by running the `sim.perf` executable via `srun` with varying numbers of threads (`--cpus-per-task=N`) on the `xs-4114`, `i7-13700`, and `w5-3423` nodes.

The raw performance data referenced in this report is available at the following link: Google Sheet.