Overview

Each timestep runs as a phased pipeline inside one OpenMP region: (1) snapshot state of cars, (2) decide lane changes using binary search and set change-lane flags for each car, (3) apply all lane flips in bulk, (4) rebuild + sort lanes, (5) update car velocity independently for each lane, then (6) advance car positions. Barriers separate dependent phases; independent loops use omp for (some with nowait) to reduce idle time.

In particular, our design includes

Algorithm:

1. Lane-change decision: Binary search functions std::lower_bound is used to search for the nearest front and back neighbours which gives O(logn) neighbor discovery.
2. Clear, rebuild in parallel: After applying lane-change flags and flipping the lane attribute of each car, both lane vectors are cleared and reconstructed from scratch, followed by a std::sort on car positions. This design avoids the costly O(n) insertions and removals that would occur if we tried to update std::vector in place. Direct insertion/removal could degrade to $O(n^2)$ in the worst case when many cars change lanes simultaneously, whereas the clear-and-rebuild strategy guarantees O(n log n) complexity from std::sort.

Data Structures:

1. std::vector lanes: two std::vector<int> lane lists store car IDs (not structs) for lane 0 and lane 1. These are rebuilt each step and kept sorted by position. Benefits: small elements (ints) make sorting and cache lines efficient; neighbor queries are by index with O(1); binary search in the other lane is O(log n)
2. Byte-sized flags (std::vector<char>):Flags for start, dec, slow-start (ss_flags), and lane-change are stored in std::vector<char> instead of std::vector<bool>. This is because std::vector<bool> is a specialized bit-packed container: multiple flags share the same word. This makes parallelism unsafe as different threads updating different flags may contend on the same word. Using std::vector<char> guarantees each flag is byte-addressable and independent, so threads can update their subset of cars in parallel without atomics or race conditions.
3. Previous-state snapshot: A copy of car states (cars_old) is written at the start of each step and used for any “last timestep” reads (like front-car velocity checks). This reduces the need for synchronization.

Parallelisation strategy

We take the simplest approach of embarrassingly parallelising work that has no dependencies. Work is divided equally among the threads using the work-sharing construct.

For each loop, four barriers are used for synchronisation. For instance, ::applyLaneChange is invoked only after ::decideLaneChange has completed. This is achieved via an implied barrier at the end of the for loop construct.

A critical region is used to synchronise thread(s) to merge work done on buffers local to thread(s).

<u>Data and Visualisation</u>

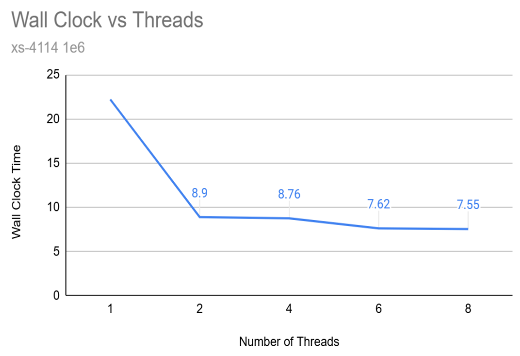


Figure 1

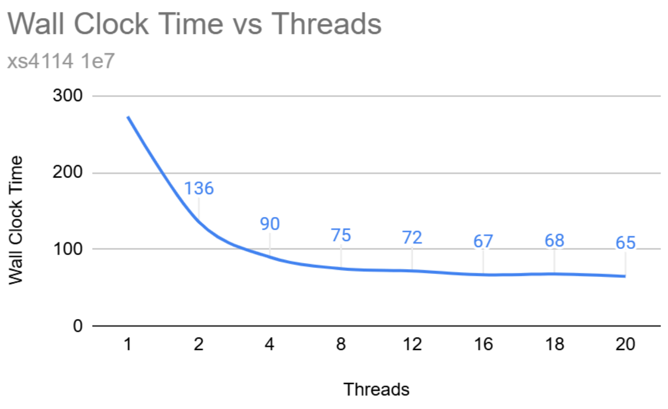


Figure 2

<u>Threads, Problem Size</u>

The program scales fairly well with the number of threads. The speedup from parallelisation decreases as the number of threads increases (Figure 1 and 2).

In addition, the program requires a large problem size of upwards of n == 1e7 to appreciate parallelism (Figure 2).

For a small problem size such as n == 1e6, speedup stagnates beyond 2 threads (Figure 1). This is due to the overhead of parallelisation outweighing the performance gain.
For a large problem size such as n == 1e7, speedup increases to 8 threads and continues to increase steadily to the maximum number of hardware threads, albeit at a slower rate. This suggests that the program is fairly scalable so long as the problem size is big enough.

Therefore, the program requires a large problem size for parallelism to be exploited. This is due to the use of for-loop constructs doing array assignment(s) where iteration space needs to be large for overhead of thread creation to be minimised.

Hardware

For a fixed problem size of moderately small size, the i7-13700 outperforms i7-7700 and xs-4114.

i7-13700 has a p/e-cores, size-able cache and fast dram. This helps to optimize parallel workloads. In comparison, the i7-7700 has a smaller cache and slower dram. This means that the i7-7700 is likely to be memory-bound when running parallel workloads. As expected, the i7-7700 speedup remains constant when threads increase from 4 and 8.

Thus, the i7-13700 has better performance than the i7-7700 on parallel workloads.

The xs-4114 has a lower clock speed and 20 cores and has good potential to handle parallel workloads. In this instance, the xs-4114 is limited by the small problem size. The wall-clock time appears to be high due to the lower clock speed.

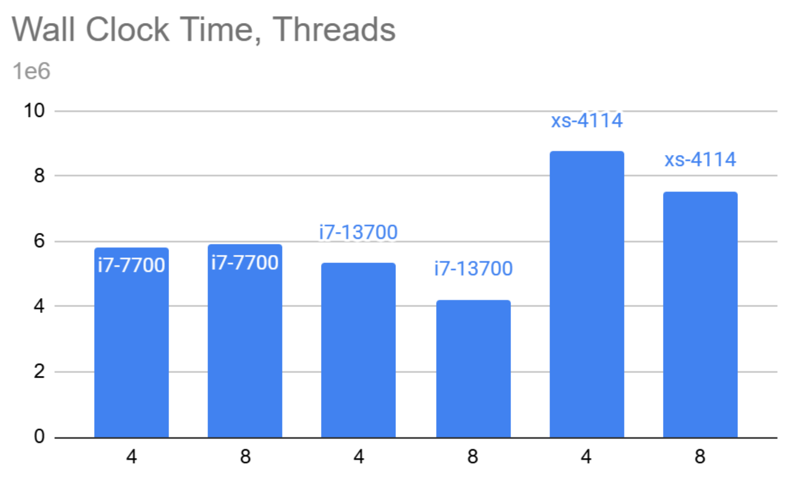


Figure 3

Performance Optimisation 1

Hypothesis: Under Rebuild and Sort Lanes, single-threaded building of arrays is bottlenecked due to sheer problem size i.e., the number of cars. There is no data dependence. Therefore, parallelizing should yield good results.

Action: In the rebuildAndSort subroutine, we use thread local buffer(s) to divide the work of rebuilding the lanes array among a team of threads.

In particular,

1. Clear once (single): One thread clears the two shared lane vectors lanes[0] and lanes[1].
2. Per-thread local partition (for): Each thread builds its own temporary vectors (thread_lane0, thread_lane1) by scanning its chunk of cars and pushing IDs to the appropriate local buffer. No shared writes happen here.
3. Batch merge (critical): Threads enter a short critical section once to append their local buffers to the shared lane vectors using insert().

This design is good as:

1. Zero contention in the hot loop: During the omp for, every push goes to a thread-local vector—no locks, no atomics, no allocator contention on shared lanes.
2. Better cache behavior: Local buffers are contiguous and private, improving locality while building the lane vectors.

The optimization gained ~ 37% speed up in wall-clock time (Figure 4) as compared to sequentially building the lanes vectors by push_back().

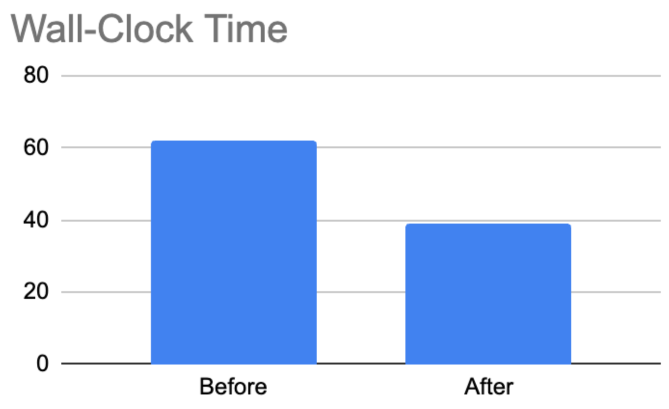


Figure 4

Performance Optimisation 2

Hypothesis: Noticing a relatively low IPC of 0.9-1.0 for moderate problem size(s), we hypothesise that the program is memory bound. Spatial locality might be sub-optimal due to small struct size of cars relative to large chunk size given to each thread. This results in false sharing of cache lines and L1 cache miss(es).

Action: We altered the data structure(s) to use a struct of arrays as opposed to an array of structs. This makes more spatial locality for memory access(es).

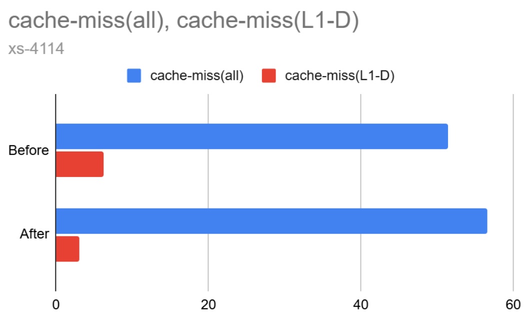


Figure 5

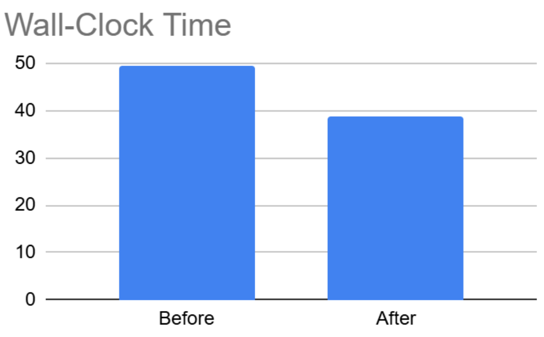


Figure 6

The optimization successfully reduced the L1-D cache miss from 6% to 3% and performed 20% faster in wall clock time. Also, the number of cache references reduced by three-fold. from 6 to 2 million, reducing cache pressure.

Appendix

Google Sheets

To test optimisation 1 and 2, refer to folder(s) opti-test-1 and opti-test-2 in github repo for documentation and relevant src code.

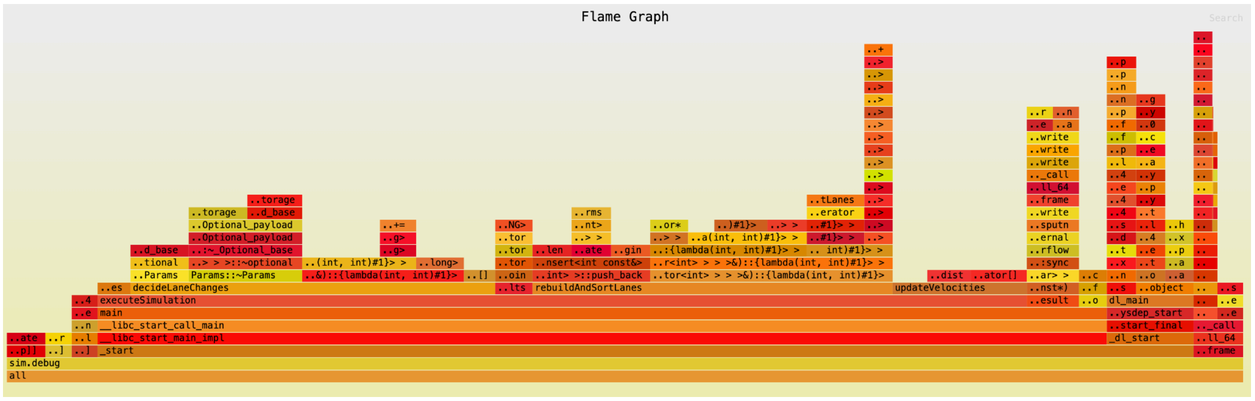
The following node(s) were used:

xs-4114 soctf-pdc-006.ddns.comp.nus.edu.sg

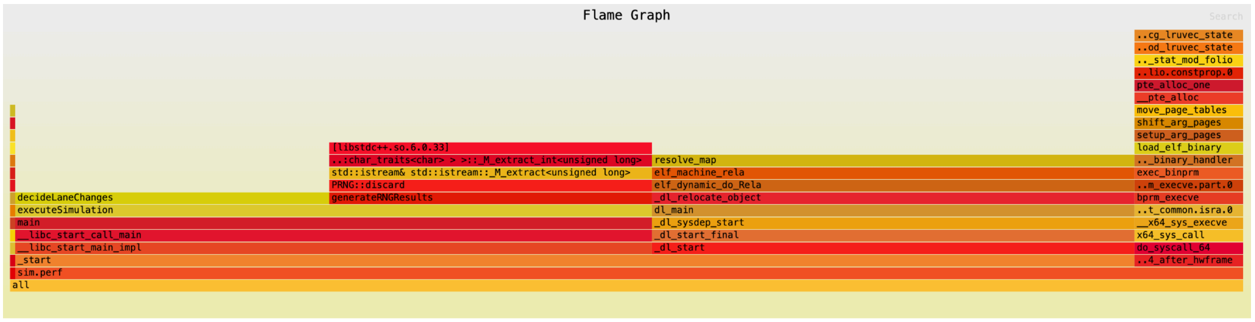
I7-7700 soctf-pdc-011.ddns.comp.nus.edu.sg

I7-13700 soctf-pdc-038.ddns.comp.nus.edu.sg

According to the flamegraph below, the workload is evenly divided among the different phase of the program with no single bottlenecks.



Parallelizing RNG generation was attempted using engine.discard unsuccessfully as discard() is the main bottleneck, making the program slower when generating RNG in parallel as compared to sequentially.



Here is the Parallel RNG program:

```
    std::vector<PRNG*> prng_engines(params.n);
    for (int id = 0; id < params.n; ++id) {
            prng_engines[id] = new PRNG(params.seed);  // All engines start with same
```

```
seed
                prng_engines[id]->discard(id * 2); // land on start of engine id's pair
        }
```

```
void generateRNGResults(Params params, std::vector<PRNG*>&prng_engines,
std::vector<uint8_t>& start, std::vector<uint8_t>& dec) {
        #pragma omp parallel for schedule(static)
        for (int id = 0; id < params.n; ++id) {
                start[id] = static_cast<uint8_t>(flip_coin(params.p_start,
prng_engines[id]));
                dec[id] = static_cast<uint8_t>(flip_coin(params.p_dec, prng_engines[id]));
                prng_engines[id]->discard((params.n - 1) * 2);
        }
}
```