Brief Description

In our implementation, one thread computes for a link or a platform. Ticks are simulated sequentially. In each tick, the program spawns trains sequentially.

Next, the program iterates through all the links in parallel and checks if the current tick is equal to or more than the completion tick (which is the tick at which the train will be done with its current activity i.e. transiting through a link or loading passengers at a platform) of the train. If it is, the train is moved to the correct holding area.

Next, the program iterates through each platform in parallel and attempts to empty every platform by moving the train occupying it to the connected link if

1. the link is unoccupied, and
2. the completion tick of the train is less than or equal to the current tick.

If the platform has been emptied or is already empty, a train will be dequeued from the platform's holding area into the platform.

The next tick will then begin. This repeats until the specified number of ticks. The program's control flow diagram can be found in Appendix A.

For link contention, we use an integer to represent the occupancy of a link. Each link will have an occupancy value tied to it. The values are:

- -2: link does not exist
- -1: link is not occupied
- >=0: id of train occupying the link

We store these integers in a 1D vector. The mapping of links to their respective index within the vector is handled by the `index_2d_to_1d` method:
`source_station_id * num_of_stations + destination_station_id`
which gives you the index of the link's occupancy value in the vector.

For data structures, we have 4 classes, station, platform, link, and train. Stations contain platforms. We store all instances of them in their respective vectors. We iterate through the link and platform vectors to perform the procedures as mentioned above. Using OMP we parallelize this by allocating some number of iterations to a thread. We also used a priority queue to handle the logic of the holding area. It first sorts by the tick the train entered then by the id of the train. Dequeuing will then give you the correct train.

For synchronisation, due to the implicit barrier of OpenMP, we do not need to add a barrier between the iterating of links and the iterating of platforms. The only synchronisation that we added manually was the critical section when pushing trains into the priority queue of the holding areas since 2 separate links might try to push 2 different trains to the same holding area at the same time, which can result in a race condition.

Speed Up Aspects

Referring to the graphs in the appendix or the full data in this spreadsheet, increasing the number of ticks or number of stations also increases the speedup in our parallel program, up to a certain point, at which it plateaus.

When the number of ticks increases, a logarithmic growth is observed for the speedup. However, the speedup decreases as the number of ticks gets larger, seemingly plateauing at 96000 ticks as shown in Chart 1.

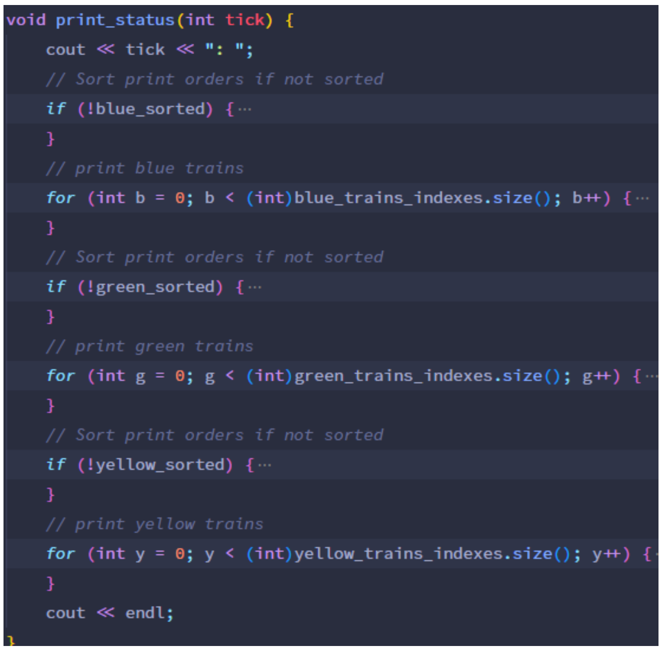
We expect that our parallel implementation computes each tick faster by a fixed speedup than the sequential program due to the usage of parallelization when given a large number of stations. However, on the contrary, there is still an increase in speedup for lower numbers of ticks, which is not within our expectations of a constant speedup. After doing measurements with perf, we believe the increase in speedup as the number of ticks increases is due to the L1 cache miss rates. As shown in Chart 2, the parallel implementation scales better in terms of L1 cache miss rates as compared to the sequential program. L1 cache miss rates for the sequential program grows much faster than the parallel implementation, which seemingly explains why there is an increase in speedup as the number of ticks increases.

When the number of stations increases, the resultant speedup increases up to roughly around 4000 stations. Since the parallel implementation does the computation for the platforms and links on every tick with the use of multiple threads, it is predicted that an increase in stations results in a logarithmic growth of computational work required for every tick, as opposed to a linear growth predicted for the sequential program.

However, when the number of stations increases beyond 4000, the speedup starts to decrease. We suspect that as the number of stations increases further, the size of the adjacency matrix to be read by the program increases exponentially, which causes the main bottleneck to be the IO. For both the provided sequential and our parallel programs, the IO time is expected to increase exponentially as the number of stations increases, to the point that most of the run time of the program is IO. This causes the computation speedup to be less significant when comparing to the overall run time of the program.

Performance Optimisation

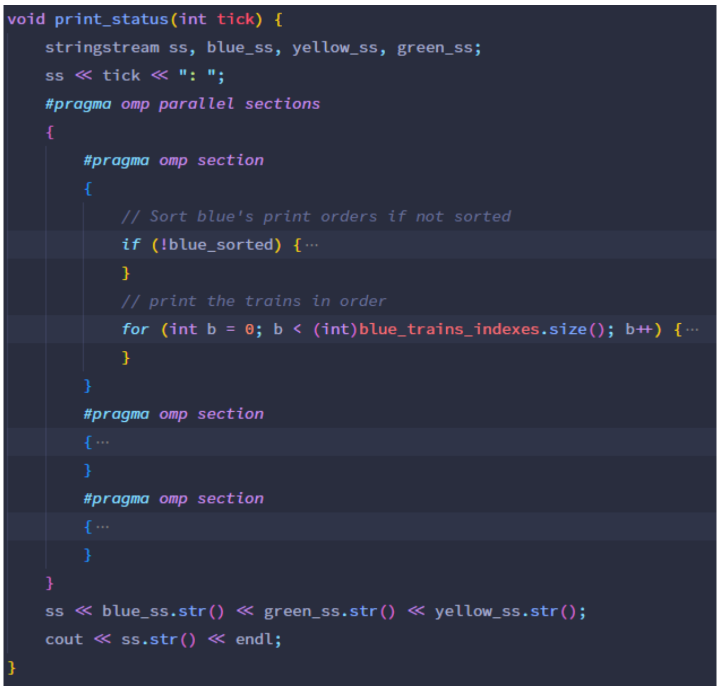
One optimisation strategy we had deployed is the parallelisation of printing of the trains for the last `NUM_LINES` ticks. Our initial implementation was to store indexes of blue, green and yellow trains on separate vectors, sort them whenever the train ids are not sorted lexicographically, and finally print them out. The snippet of the code can be seen below in Figure 1, or within the seq_print branch in the repository.



```
void print_status(int tick) {
    cout << tick << ": ";
    // Sort print orders if not sorted
    if (!blue_sorted) {…
    }
    // print blue trains
    for (int b = 0; b < (int)blue_trains_indexes.size(); b++) {…
    }
    // Sort print orders if not sorted
    if (!green_sorted) {…
    }
    // print green trains
    for (int g = 0; g < (int)green_trains_indexes.size(); g++) {…
    }
    // Sort print orders if not sorted
    if (!yellow_sorted) {…
    }
    // print yellow trains
    for (int y = 0; y < (int)yellow_trains_indexes.size(); y++) {
    }
    cout << endl;
}
```

Figure 1: sequential `print_status()` code

Later on, we had identified that the printing is an area of improvement, and can be optimised with the use of OpenMP parallel sections. Instead of sorting and printing the trains directly to stdout sequentially, we declare a new stringstream for each line colour, and define an OpenMP parallel section for each colour so that the sorting and building of the building of the string to be printed can happen in parallel. The snippet of the final parallelised code is depicted in Figure 2 below.

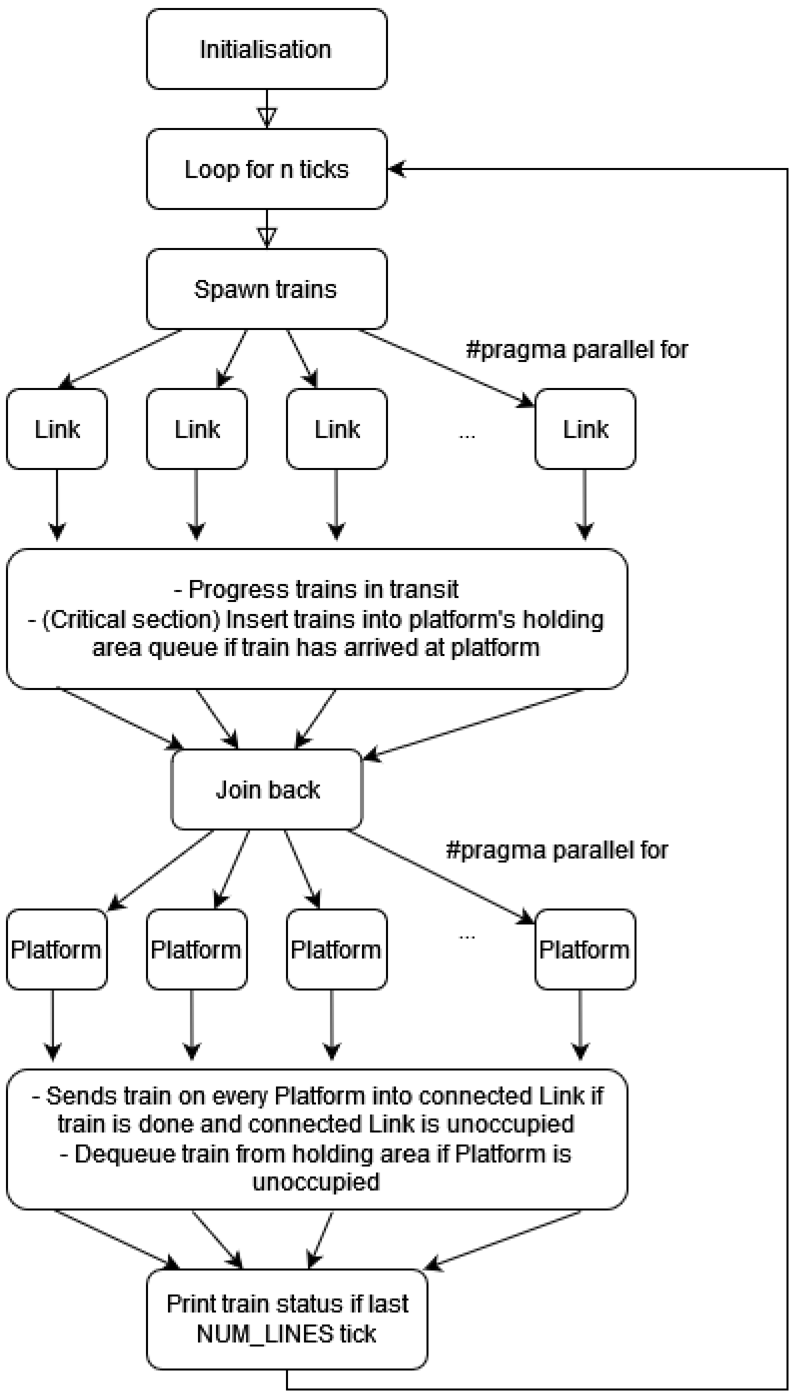


```
void print_status(int tick) {
    stringstream ss, blue_ss, yellow_ss, green_ss;
    ss << tick << ": ";
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            // Sort blue's print orders if not sorted
            if (!blue_sorted) {···
            }
            // print the trains in order
            for (int b = 0; b < (int)blue_trains_indexes.size(); b++) {···
            }
        }
        #pragma omp section
        {···
        }
        #pragma omp section
        {···
        }
    }
    ss << blue_ss.str() << green_ss.str() << yellow_ss.str();
    cout << ss.str() << endl;
}
```

Figure 2: parallel `print_status()` code

According to our measurements as depicted in Charts 4 and 5, for a constant number of stations, ticks to simulate and ticks to be printed, the parallelisation of `print_status()` offers a slight speedup for small numbers of trains, but offers up to 1.25 speedup compared to the sequential `print_status()` for large numbers of trains over 6000 trains. The full measurement data can be viewed in a spreadsheet here. As observed, there is a plateau in the speedup as we can only make use of at most 3 threads to parallelise the sorting and printing. If our program is required to compute for more train line colours, the speedup of the sorting and printing will be more significant as more threads can be utilised to help with the parallelised task.

Appendix A: Program Control Flow Diagram



Initialisation
Loop for n ticks
Spawn trains
#pragma parallel for
Link
Link
Link
...
Link
- Progress trains in transit
- (Critical section) Insert trains into platform's holding area queue if train has arrived at platform
Join back
#pragma parallel for
Platform
Platform
Platform
...
Platform
- Sends train on every Platform into connected Link if train is done and connected Link is unoccupied
- Dequeue train from holding area if Platform is unoccupied
Print train status if last NUM_LINES tick

Appendix B: Chart results

Chart 1 (i7-9700)

3500 Stations 120000 Trains

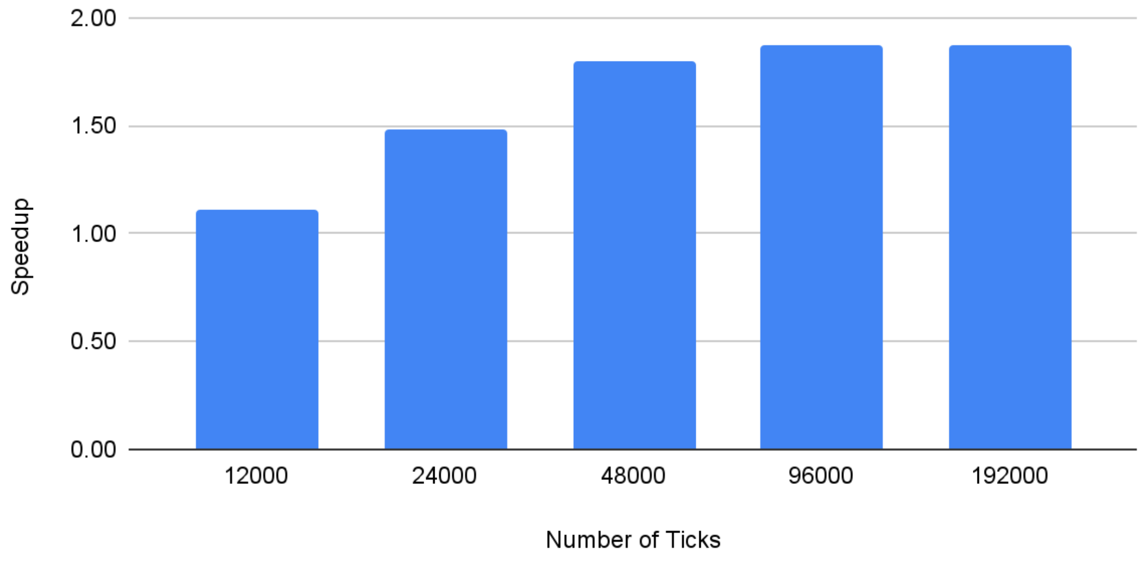


Chart 2 (i7-9700)

Cache Miss Rates against Number of Ticks

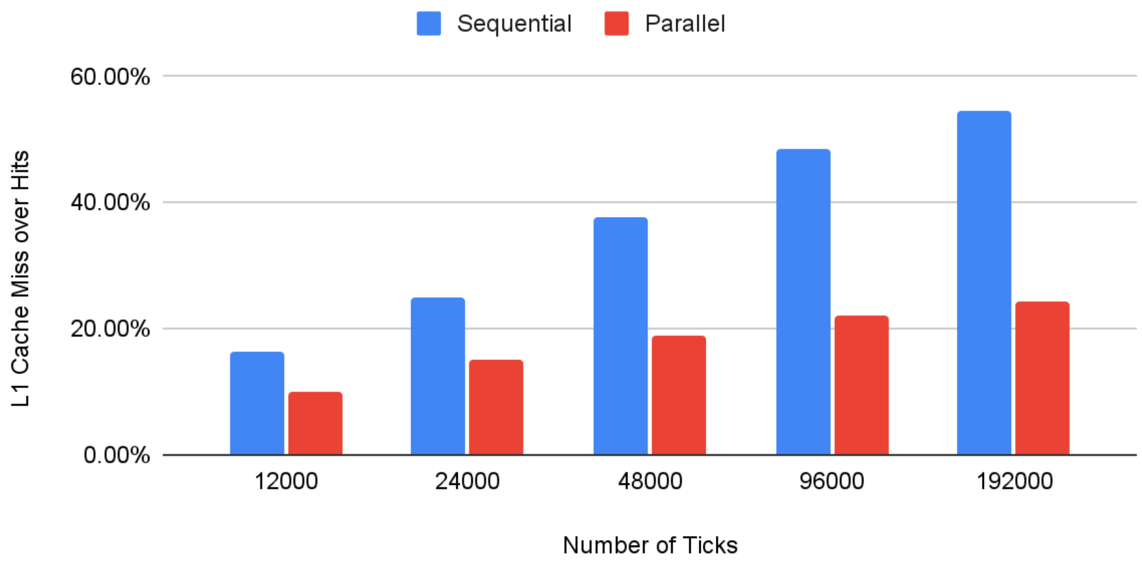


Chart 3 (Xeon Silver 4114)

3000 Trains 24000 Ticks

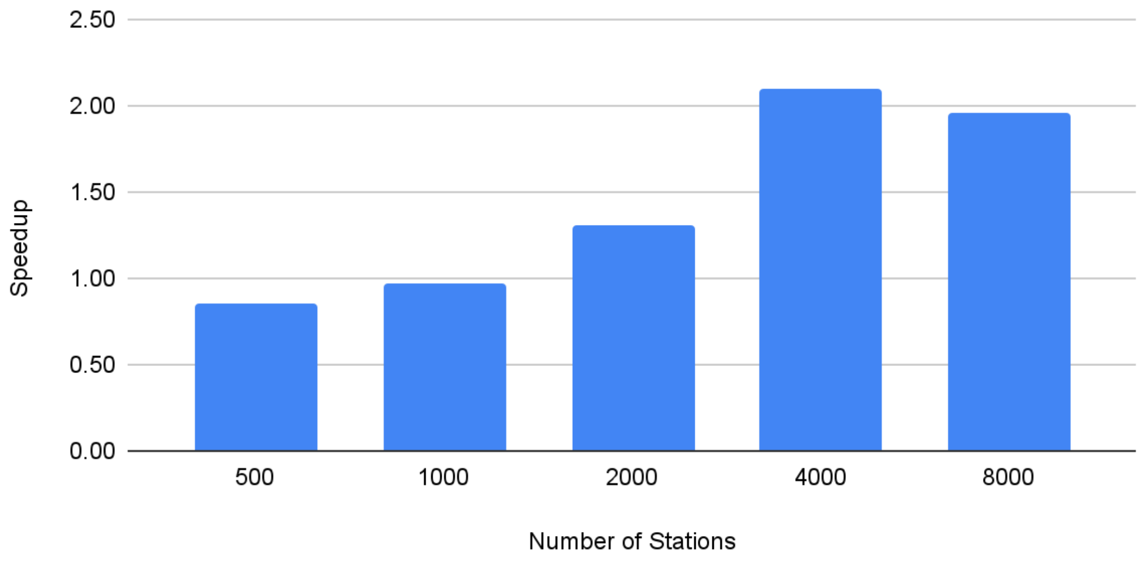
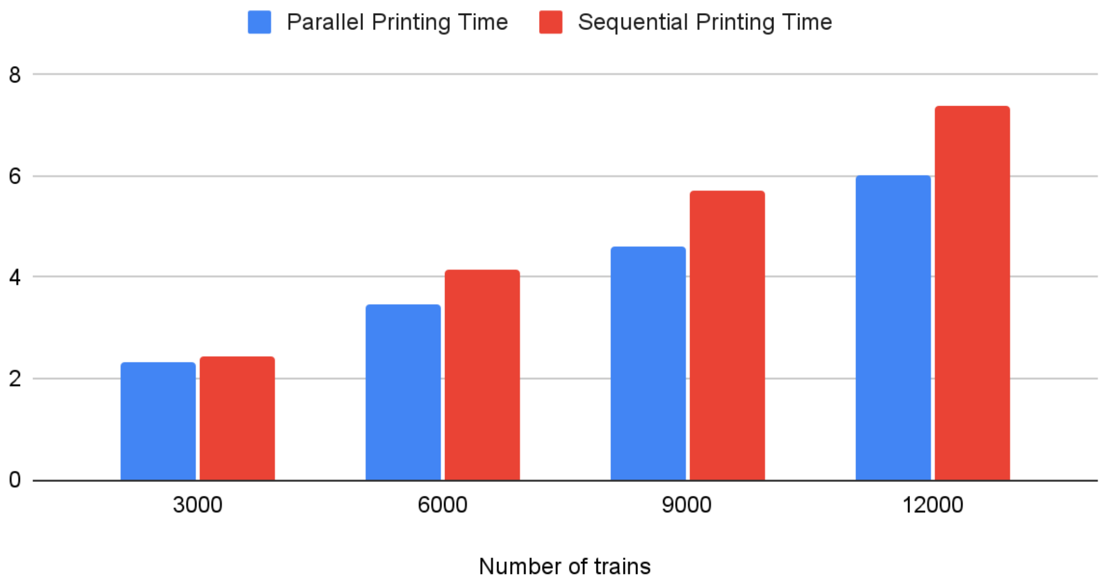
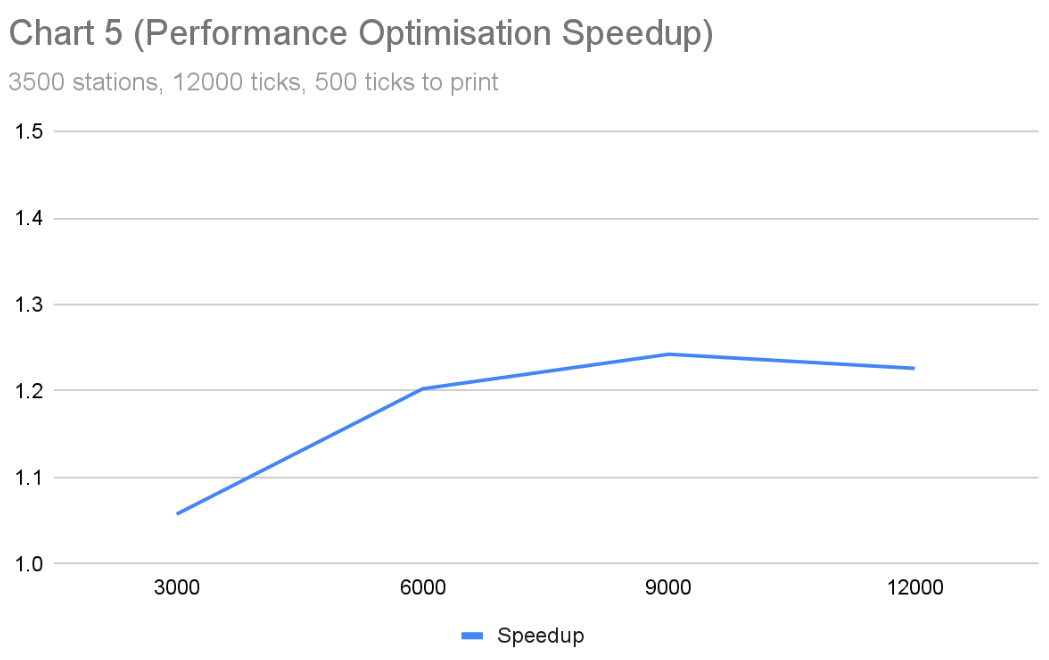


Chart 4 (Performance Optimisation on Xeon Silver 4114)

3500 stations, 12000 ticks, 500 ticks to print





Appendix C: Reproduction of Results

For all the charts, we use the following perf command within job.sh:

```
perf stat -r 5 -e
L1-dcache-load,
L1-dcache-load-misses,
cache-references,
cache-misses,
cycles,
instructions
```

For Charts 1, 2, 4 & 5

Using the `own_sample_3500.in` file in `testcases/` in the repo, change the number of trains (divided amongst 3 lines), number of ticks, and number of prints in the file accordingly.

For Chart 3

Copy the `own_sampleX.in` files in `testcases/station_count_comparison/` within the repo to `code/` folder. `X` stands for the number of stations in the input file.

Modified `run_job.sh` script is on
https://gist.github.com/Polygonalr/fcce6cac11bf4d10826e1a7445a35837

Usage: `./run_job.sh <executable_name> <no of trains>`, the output is created in the same directory with the filename `<executable_name>_<no_of_trains>.out`.