Challenge 2: Rush Hour

Rush hour is a puzzle game that is also a popular smartphone game called *Unblock me*.

https://en.wikipedia.org/wiki/Rush_Hour_(puzzle)

Problem

Several cars are jammed in an intersection. The goal is to get a special car from one side of the playing field to the other. Each car has a fixed orientation and can either move forward or backward. The car can only move if it is not blocked by another car or by the playing field's borders.

To simplify programming this task, you are provided with the following set of classes and methods that manage the game board and movements of the car:

Classes

- `Car`: Models a single car.
- `State`: Models a configuration of cars.
- `StateManager`: Keeps track of the current best solution and the states that have already been checked.

Functions

- `Check(...)`: checks whether a state fulfills the game's goal, recursively creates and checks follow-up configurations.
- `main(...)`: Sets up the initial state and calls `Check(...)` on it.

Along with the code template, two initial configurations *sample.json* and *main.json* can be found in the *configs* directory. The *sample.json* is a shorter one that can easily be solved using pen and paper to understand the problem better.

The following set of tasks can be used as initial steps to solve the problem.

Task 0

Draw the initial configuration given in the `sample.json` using pen and paper. The first car in the configuration is the one that needs to reach the goal side of the playing field. Work out the shortest solution to compare your algorithm against. The length of a solution is determined by the number of one-field moves done by the cars.

Task 1

Implement the body of the `Check(...)` function. It should perform the following operations, these also provided within the code template:

- Immediately return if the State to be analyzed took more steps to reach than our current best solution.
- Try to claim its state in the manager. If the state is already claimed by another task, immediately return.
- Check if the state is a winning state, if so, enter its solution into the manager and return.

- Iterate over all the cars (`state.carCount()`). For each car create the two follow-up states created by moving the respective car forward or backward. `state.move_car(...)` returns such a follow-up state from a given car number and direction.
- Check whether the follow-up states created are legal states. If so, recursively call `Check(...)` on them.

Compile the code. For example:

```
g++ -Wall -fopenmp Car.cpp State.cpp StateManager.cpp
    RushHour.cpp -o RushHour
```

Execute the code:

```
./RushHour <configuration>
./RushHour configs/main.json
```

Compare the solution obtained with what you worked out in Task 0.

Challenge

Write a program that uses OpenMP to parallelize the Rushhour problem to obtain the best speedup possible.

Execute the final solution with the `main.json` configuration and output the wall time of the execution.

Rules of the Challenge

- The challenge can be attempted as a single participant.
- Top 5 teams that win the challenge should be prepared to present their solutions.
- The submitted code will be tested with other initial configurations.

Submission

1. Final Date of submission is *26/01/2021* at *23:59*.
2. Matriculation number and full names of every participant should be mentioned at the header of every file submitted.
3. Files have to be submitted in Panda in the respective challenge sub-section within the Practical Lab section.
4. Files to be submitted:
   a) Batch script used to schedule your job in Noctua.
   b) Every file already provided and all new files created for the program.

Hint

`StateManager` is responsible for the state that is shared across all tasks, yet it lacks any precautions against data races. Add a Mutex (`omp_lock_t` from `<omp.h>` or `std::Mutex` from `<mutex>`) and use it where necessary to make the program safe.