Low-Level Parallel Programming

Assignment 3
Deadline: February 26 23:59

Uppsala University, Spring 2021

1 Deadlines, Demonstrations, Submissions, and Bonus Points

Each lab report and source codes zipped together has to be submitted by its corresponding deadline. During the lab session on the same day, demonstrate your working solution to a lab assistant. Be prepared to answer questions.

For this lab, there is an opportunity to obtain a bonus point. Bonus points can be accumulated throughout the rest of the course. One bonus point will raise your final course grade from 3 to 4, while two bonus points will raise it to 5. There will be one more opportunity to get bonus points after this lab. Please remember to divide the work equally within the group!

2 Lab Instructions

The objective of this lab is to get an insight into load balancing, methods for race-free inter-thread interaction, and basic task parallelism.

You may have noticed that the agents move straight through each other, without acknowledging each other's existence in any other way than turning red. This is naturally unacceptable behaviour, and thus we will fix this during this lab. In the given files there is an implementation for collision prevention. An agent will move to any of three different locations that brings it closer to its destination (the desired position, and the two neighbouring positions closest to that), **if they are empty**. It prioritizes to the *desired* one. [1]

For this lab, you may choose to use either threads, openMP, CUDA, or a combination thereof. Pick the framework you feel is most suitable for the job[2].

Step 1: Modify your code so that it uses the function *Ped::Model::move* to set the correct x and y position for each agent. See the code listing below for how to do this for the sequential case. Make sure that all versions (sequential, omp ...) from previous assignments are still functional.

```
for(agent: agents){
    agent->computeNextDesiredPosition();

    // Replace this (or your equivalent code)
    // agent->setX(agent->getDesiredX());
    // agent->setY(agent->getDesiredY());

    // with this:
    move(agent);
}
```

The collision prevention logic is an additional step after the *computeNextDesiredPosition* function.

Step 2: Parallelize the collision prevention logic.

[1] This is actually not a very good heuristic, as you may notice. Feel free to play around with it and try to improve it. A good addition is to try to back off instead of just staying put.
[2] Hint: Do not pick CUDA.

After some examination, you will find that the code can not be trivially parallelized. The problem is not in the structure, as in assignment 2, but rather in the algorithm itself. Even though two agents may desire the same location, **only one agent may move to each location per tick, and that location must be otherwise empty.** When agents are handled in different threads, care must be taken so that *race conditions* do not occur.

Step 3: Evaluate the quality of your solution.

2.1 A Simple Solution

You should **not** implement this solution, only consider its efficiency.

Consider each 2d location as a resource, which only one thread may change/use at any given time. Resources can be acquired in a thread safe manner by using locks. One solution is to use a single global lock. All threads that are about to move an agent acquires the lock, double checks that the location is empty, moves, and releases the lock. A different approach is to guard every single location with a separate lock. This will churn out a lot of locks, but some optimizations are possible.

2.2 Parallelize Using Regions

(This is the solution you should implement for this lab)

Divide the 2d world into regions, and treat each region as a task. You may keep the number of regions constant, and different from the number of available threads. However, the number of regions must be at least four. All agents with coordinates within the bounds of a region belong to it. All movements within the region can now be safely made without any locks!

When an agent moves into another region, the ownership of that agent must be transferred to the new task. Additionally, agents moving along and/or across the borders may experience race conditions. You need to incorporate a solution for these events in your code. **The final solution must be guaranteed to be free of race conditions, and the collision prevention must never fail.** We do not, however, require that the outcomes of the competition of locations are completely "fair".

As before, it must still be possible to choose between the versions implemented in previous assignments. Independently of this, it should also be able to choose between the sequential and parallel version of the new code.

2.3 Bonus Point Assignment

For the bonus point there are additional requirements for your solution.

- It must be possible to automatically adjust the number of regions at each tick. To improve load-balancing, you can split heavily populated regions, as well as merge sparsely populated regions.
- Your implementation may neither use locks, nor contain critical sections. You may want to use *CAS*[3].

2.3.1 Bonus Assignment Questions

A. Briefly describe your solution. Focus on how you addressed load balancing and synchronization, and justify your design decisions.

B. Can your solution be improved? Describe changes you can make, and how it would affect the execution time.

[3] Compare And Swap

2.4 Evaluation

For the timing evaluation, choose one of your implementations of *computeNextDesiredPosition*, and use it for all measurements. Compare the execution time when swapping the sequential version of the collision free movement to your parallelization, using a few interesting scenarios of your choice. Run each experiment several times. Use the mean values to plot and compare performance. Remember to use the metrics(s) that best describe what you want to present when you decide how to plot your results.

2.5 Questions

A. Is it possible to parallelize the given code with a simple *omp parallel for* (compare assignment 1)? Explain.

B. How would the global lock solution (sec 2.1) perform with increasing number of threads? Would the second simple solution perform better in this regard? Can the scenario affect the relative performance of the second simple solution?

C. Consider a scenario file designed to generate the worst-case load balancing for your solution. Describe and try to quantify how bad it could get. Worse than the sequential solution?

D. Would your solution scale on a 100+ core machine?

3 Submission Instructions

Submit the code and a **short** report including all answers to above questions in **PDF** format to studium.

1. Your code **has** to compile.
2. **Clean** your project (run **make submission** which will copy code files from demo and libpedsim and make a tarball called submission.tar.gz.
3. Document how to choose different versions (serial, C++ Threads, and OpenMP implementation).
4. Include a specification of the machine you were running your experiments on.
5. State the question and task number.
6. State on the assignment whether all group members have contributed equally towards the solution - it will be important for your final grade.
7. Include all generated plots.
8. Put everything into a zip file and name it team-n-lastname$_1$-lastname$_2$-lastname$_3$.zip, where n is your team number.
9. Upload the zip file on studium by the corresponding deadline.

4 Acknowledgment

This project includes software from the PEDSIM[4] simulator, a microscopic pedestrian crowd simulation system, that was originally adapted for the Low-Level Parallel Programming course 2015 at Uppsala University.

[4] http://pedsim.silmaril.org/