Low-Level Parallel Programming

Lab Assignment 3
Lab 3 Deadline: February 25 23:59

1 Deadlines, Demonstrations, Submissions, and Bonus Points

Each lab has to be submitted by its corresponding deadline. During the following lab session, 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 at least two more opportunities to get bonus points after this lab. Please remember to divide 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. 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. The implementation uses a quadtree for fast neighbour search. An agent can move to three different locations that brings it closer to its destination (the desired position, and the two neighboring positions closest to that), if they are empty. It prioritizes to the *desired* one.

For this lab, you may choose to use either pthreads, openMP, CUDA, or a combination thereof. Pick the framework you feel is most suitable for the job[1].

[1] Hint: Do not pick CUDA.

Step 1: Patch in the collision prevention logic into your code base. If you have followed the instructions from lab 2 well, then it should be possible to do with little effort. If not, you are allowed to get creative.

The collision prevention logic is an additional step after the *go* function. We provide you with the sequential version of the collision handling. The changes affect p*ed_model.h*, *ped_model.cpp*, *ped_agent.h*, *ped_agent.cpp*. The changes are summarized below.

- Each agent has a position and a desiredPosition (the old position).
- Each agent has corresponding getters and setters.
- The method go() now computes the desiredPosition (you may stick with this name for simplicity).
- After go(), collision is handled and the agent's position is set.
- libpedsim contains a new data structure, the *quadtree*.

Step 2: Parallelize the collision prevention logic.

After some examination, you will find that the code can not be trivially parallelized. The problem is not in the structure, as in lab 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.

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 world into a 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 within the bounding

box of a rectangle belongs to that region. 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 combinations of implementations for whereToGo, and go. It should also be able to choose between the sequential and parallel version of the new code, independently of the chosen whereToGo- and go implementation.

2.3 Bonus Point Assignment

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

- It must be possible to automatically adjust the number of regions at each tick. To improve load-balancing, you can split heavy populated regions, as well as merge sparsely populated regions.
- Implement a lock-free implementation. You may want to use CAS^2 .

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.

2.4 Evaluation

For the timing evaluation, choose one of your implementations of whereToGo, Go, and use them for all measurements. Compare the execution time when swapping the sequential version of the collision free movement to your parallellization, using a few interesting scenarios of your choice. Run each experiment several times and use the average for plotting.

[2] Compare And Swap

2.5 Questions

A. Is it possible to parallelize the given code with a simple *omp parallel for* (compare lab 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. Is the workload distributed evenly across the threads?

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

3 Submission Instructions

Submit the code and a **short** report including all answers in **PDF** format to student-portalen.

1. State in the report the contribution of each group member in percentage (if all members contributed equally, then write that).
2. Your code **has** to compile.
3. Document how to choose different versions, if more than one.
4. Include a specification of the machine you were running your experiments on.
5. State the question and task number.
6. Put everything into a directory and create a zip file and name it team-n-lastname$_1$-lastname$_2$-lastname$_3$.zip, where n is your team number.
7. Upload the zip file on Studentportalen by the corresponding deadline.

4 Acknowledgment

This project includes software from the PEDSIM[3] simulator, a microscopic pedestrian crowd simulation system, that was adapted for the Low-Level Parallel Programming course 2015 at Uppsala University.

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