Low-Level Parallel Programming (1DL550) Report for assignment 3 by group 4

Tommy Vågbratt(33.3%) Daniel Edin(33.3%) Lucas Arnström(33.3%)

March 8, 2016

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

It would not work to only use a single omp parallel pragma to parallelize the simulation because there would be a several data races when the agents move. The collision detection has a dependency on nearby agents which will fail.

2 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?

It would perform worse and worse. Having a single global lock would essentially make the solution sequential since all modifications of the agents locations would be performed one at a time. The only computation performed in parallel would be the calculations of their desired positions.

The second solution would perform better since all modifications of the agents locations would be made in parallel. The drawback would be the overhead of acquiring and releasing mutex locks for each update, as well as storing all locks.

3 Is the workload distributed evenly across the threads?

The world will be divided into multiple smaller regions and each region will be treated as a task by OpenMP. Which allows OpenMP to parallelize better since there will be more smaller tasks instead of a few big tasks if the amount of regions is fixed.

4 Would your solution scale on a 100+ core machine?

Updating all agents contained within a single region is a task and all tasks are distributed over the number of threads. The world can be divided into 100+ regions so it will scale to a certain degree. How well it scales will depend on the scenario file which includes the amount of agents and how the waypoints are set up.

5 Bonus Assignment

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

The biggest difference in comparison with the original code is that we do not use the provided quad tree to find nearby agents for the collision. Instead a matrix with atomic booleans (from the standard library) is used to mark where all agents are positioned. This matrix allows us to directly check each agent's desired position if it is taken. In comparison with traversing the quad tree to find all nearby agents and then match their positions.

For load balancing our solution can split populated regions in half. Regions can also merge with their parent region when needed. The merging is weighted to minimize unnecessary merging.

If an agent moves to a region's edge, compare-and-swap is used on the matrix to eliminate data races over regions. Movements inside each region is done without the compare-and-swap instruction.

Each region holds a thread safe stack with agents that entered its bounds the previous tick. The stack is emptied safely by the region in the start of each tick before any movement is done. This thread safe stack allows us to move agents from multiple other regions in to one region in parallel without data races.

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

By having each region have their own local matrix, the dreadful false sharing of the current matrix can be diminished. A global but much less used data structure to use with compare-and-swap will still be needed.

Currently when an agent is moved to a new region a search through all regions is made to find the target region. This could be improved by either for each region to store pointers to all their neighbouring regions or store the regions in a quad tree which to speed up the search.

6 How to choose different implementations

To run the code for this assignment use `--collision` flag and use `--implementation=<seq|parallel>` to change between the sequential and parallel implementation. Use `--color=<none|region|cross>` to enable color mode. See below

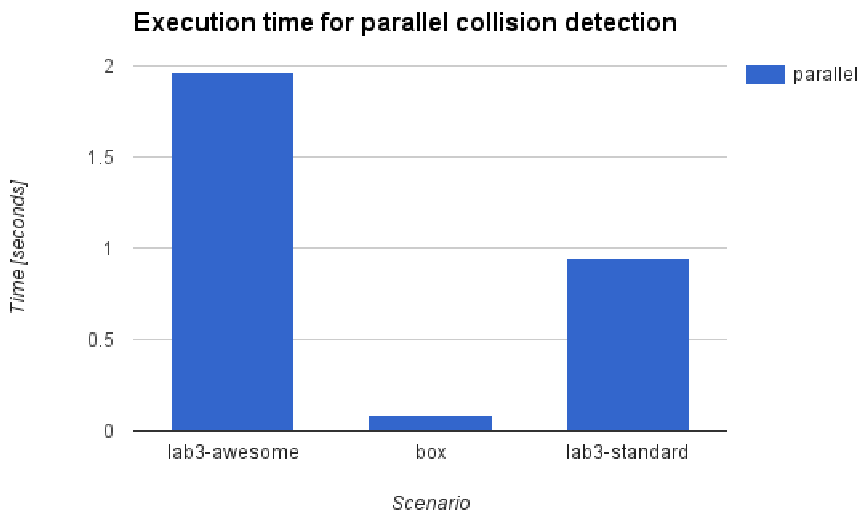

Figure 1: The average execution time for the parallel implementation of the collision detection. The scenarios used was our own “lab3-awesome” scenario, the “box” scenario, and the original “lab3” scenario. From left to right 1.96234, 0.08504, 0.947.

```
./demo [--help] [--timing-mode]
       [--implementation=<seq|omp|pthread|cuda|vector|parallel>]
       [--nthreads num] [--collision] [--color=<none|cross|region>] [scenario]

        --collision   can only be used with --implementation=<seq|parallel>
                      defaults to --implementation=seq
```

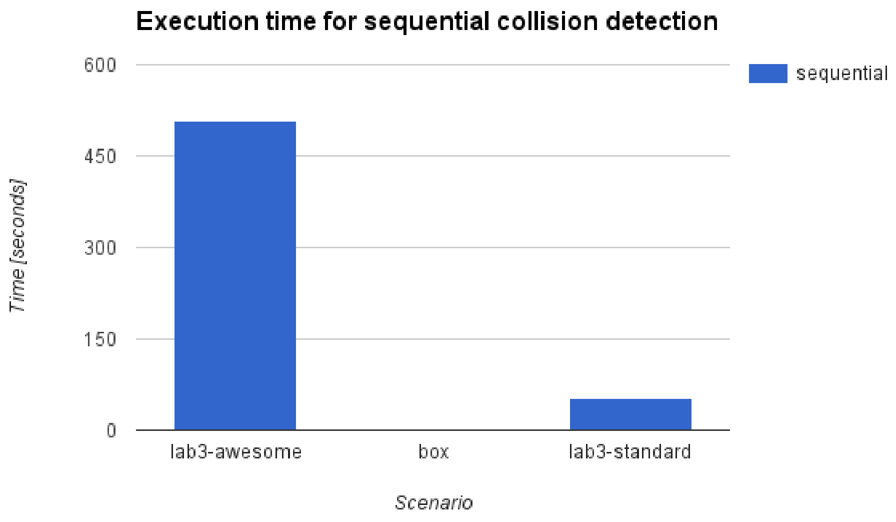

Figure 2: The average execution time for the sequential implementation of the collision detection. The scenarios used where the same as for figure 1. From left to right 508.129, 0.39048, 53.2597.