Low-Level Parallel Programming Uppsala University – Spring 2015 Report for Lab 1 by Team 21

Markus Palacios, Jonathan Sharyari, Peng Sun

Note:
We chose to create two implementations of this: one with pthreads and regions, and one simple omp-pragma. This was made possible by removing the tree-infrastructure, replacing it with a simple 2d-grid of booleans, indicating whether a certain position is occupied or not, making changes with CAS.

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

No. Since each movement of agents can affect others' decision. If a simple omp parallel for is implemented, there might be several agents that move to the same position. The tree implementation must be modified.

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?

If we increased the number of threads, we will get the same performance as sequential in the best case, due to the fact that other threads must wait for the lock to be unlocked by the thread currently executing. Likely, the overhead would be huge.

The second simple solution will perform better, because the lock will only be set on the agents that move cross regions or alongside borders. For the movements inside each region there will not be a global lock that blocks other threads.

The scenario can affect the relative performance. The fewer agents move cross regions or alongside borders, the fewer global locks will be involved, the better performance will be get.

C. Is the workload distributed evenly across the threads?
Not exactly. In our implementation we try to distribute number of agents "evenly" to each thread, creating boxes dynamically. But the number of agents that move cross regions or alongside borders cannot be predicted. So each thread will not have the same amount of work.

Regarding the openMP implementation, this can be done using dynamic scheduling.

D. Would your solution scale on a 100+ core machine?
With some modifications (number of threads) our solution should scale on a 100+ core machine.
Both implementations would scale, due to the simplicity of the solution, provided the simulations are scaling as well (i.e. 1 agent per core would be a waste of resources).

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

We implement a grid that keeps the record if a position is occupied by an agent. And we divide the whole world horizontally into 4 regions (can be adjusted). For each tick we sort all the agents' x-coordinates and pick up the median. Then we can find a fairly good point to divide agents evenly. This will guarantee the load balance – each threads will almost have the same number for agents to deal with.

For each desired position of agents we get, we perform a CAS guard, which will eliminate racing condition. The agent will only move to a point that is not occupied by other agents or assigned to other agents' desired positions.

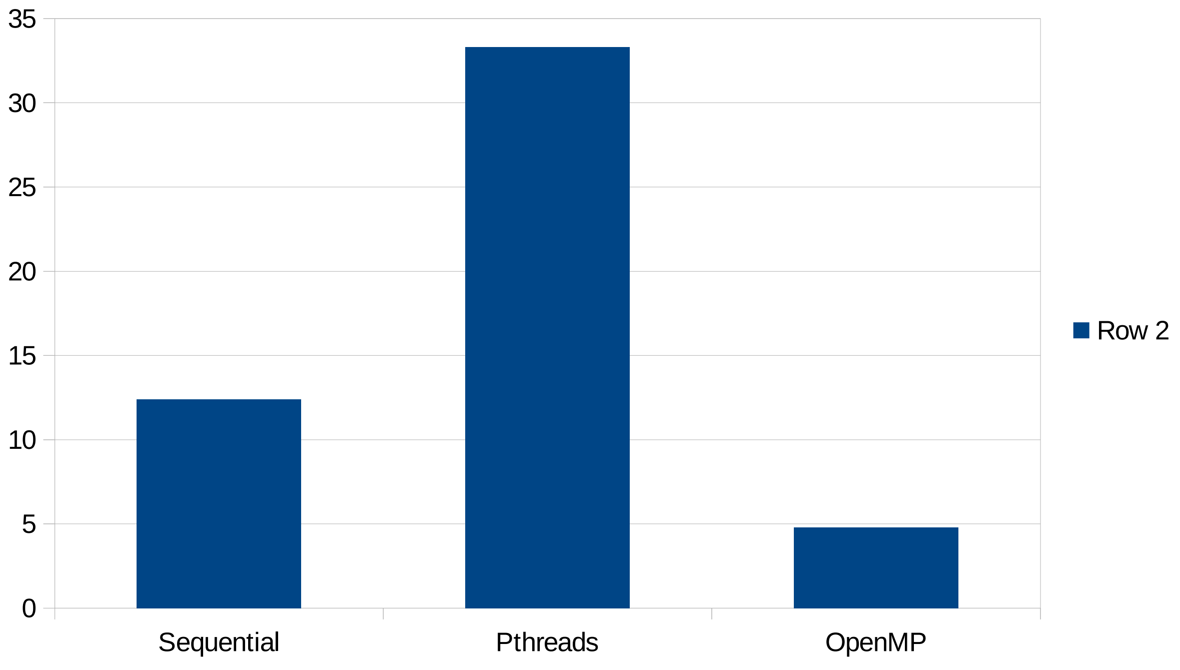
B. Can your solution be improved? Describe changes you can make, and how it would affect the execution time

It might be possible to improve our OpenMP solution by introducing a dynamic scheduling with OpenMP. This will improve load balancing for each thread and possibly shorten the execution time.

The pthreads version is far from optimal – the main reason was to implement a region-based solution. The time spent on creating these regions is simply a waste of time, as collisions are avoided in any case.

Experiments
The results of our experiments can be seen in the graph below. The experiments were run with 2048 agents (before removing duplicates), and shows the average of 3 runs. The pthreads version is a lot slower than sequential, but one must bare in mind that the sequential version without trees is already much faster than the original version with trees.

Our experiments were run under Ubuntu Linux ~14.04 (64~bit) on an Intel Core~i3~550 of 3.2~GHz with an 4~MB L2 cache and a 4~GB RAM.



Run instructions
Without an argument specifying the type of parallelism, the serial version is used, whereas --pthreads, --openmp, --vector and --cuda activate pthreads, openmp, vector and cuda respectively (in case both are set, the last occurance will be dominating). The number of threads can in both cases be set by --np X. The value of X is ignored in the serial case. Timing mode is enabled by adding --timing-mode to the commandline. An xml-formatted scenario may be specified by writing the name of the xml file in the command-line, if none is mentioned, "scenario.xml" will be used.

If no collision detection method is specified, pthreads is assumed. For sequential, use --cseq and --comp for openMP. Note that openMP requires that the number of threads is specified.

```
Serial: ./demo
OpenMP: ./demo --openmp
Pthreads: ./demo --pthreads --np 4
Vector ./demo --timing-mode –-vector
CUDA ./demo --cuda hugeScenario.xml
```

Other
We certify that this report is solely produced by us, except where explicitly stated otherwise and clearly referenced, and that we can individually explain any part of it at the moment of submitting this report. We believe to have contributed equally to this lab, within reasonable bounds.