APD Project 1: Epidemics

Brebu Iasmin-Marian, 3C.1.1

Overview

The population is distributed equally among each thread, except the last which also has the remainder which could not be evenly distributed, and for each iteration a thread:

- Updates the position of its population
- If a person is infected, then it marks the position as being infected in contagion zone
- Waits for all other threads to update the position of their population (barrier)
- Determines the status of each person based on its current status and if it is in a contagious zone
- Waits for every other thread to finish their status update (barrier)
- Cleans the are of the contagious zone that is has been assigned to it
- Waits for every other thread to finish cleaning before starting a new iteration (barrier)

The serial version is conceptually the same as if we only had 1 thread.

There are 2 modes of operation: debug or no debug. Debug is done by defining DEBUG (#define DEBUG), and no debug by not defining it ().

In debug mode the status of each person is printed on every iteration together with the contagion zone.

In no debug only the measured times together with the speedup and the result verification is printed.

The verification is done simply by comparing the resulting array from the parallel and serial functions.

Determining the next status

Here there were a few possible options I thought about:

1. For each person iterating over every infected person to determine if any of them share the same coordinate.

2. An update from option 1 would be for each grid position to hold lists of its populations: infected, susceptible, immune. And then for each susceptible iterate the infected list.
3. Mark down each spot with an infected person and then simply check that spot. Firstly

I choose to go with the 3rd option as it was relatively simple and it's performance was quite good: O(1) to determine if a person needs to be infected, and it could be easily updated when calculating the position.

Running the measurements

To compile there is a simple Makefile that manages that.

For running multiple experiments and logging them, there is a run.sh file that runs the program with multiple parameter combinations. The result is in results.txt and it isn't pretty, to be useable we need to modify the program to not write anything except csv format (comment all prints, uncomment the ones that are dealing with csv).

A cleaned-up results batch together with graphs is in apdGraphs.xlsx.

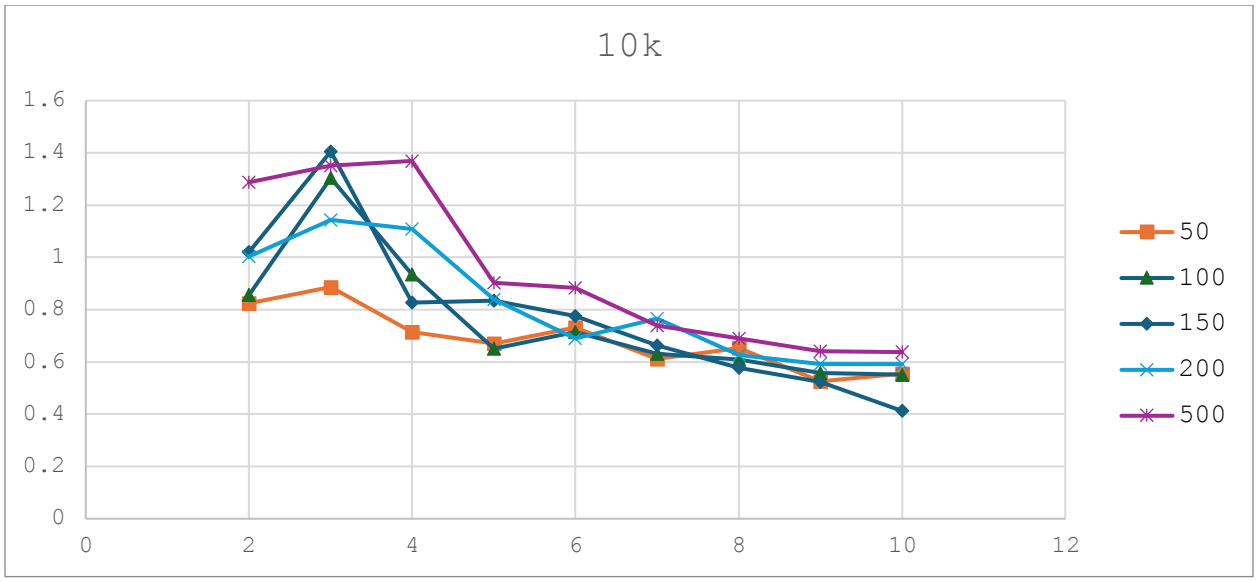
Measurements results

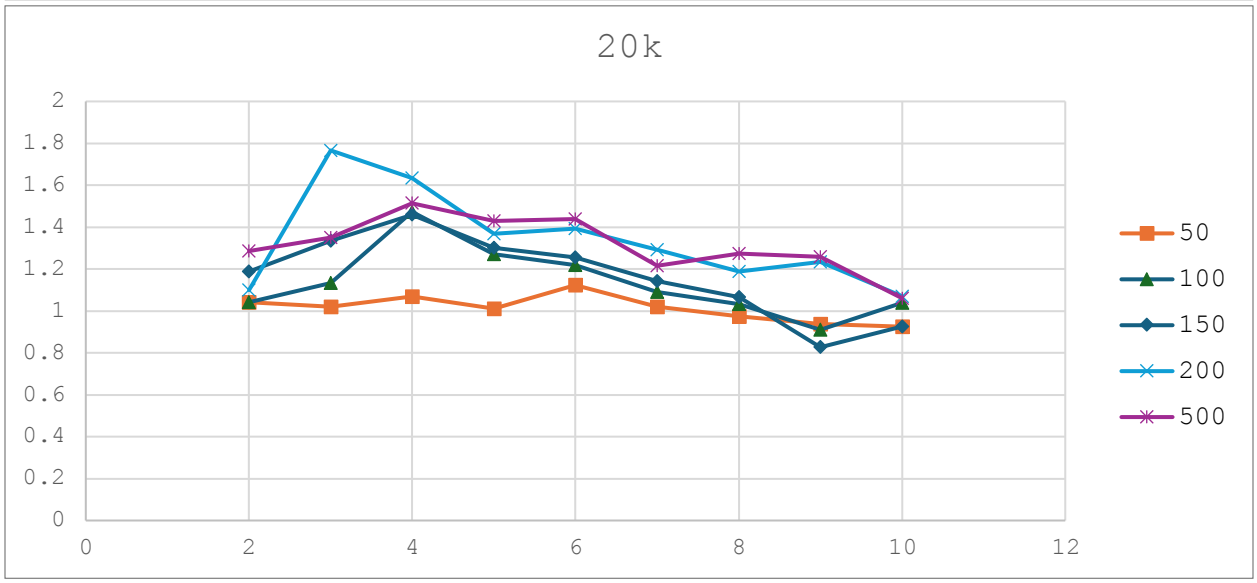
The measurements have been done doing all possible combinations of the following parameters:

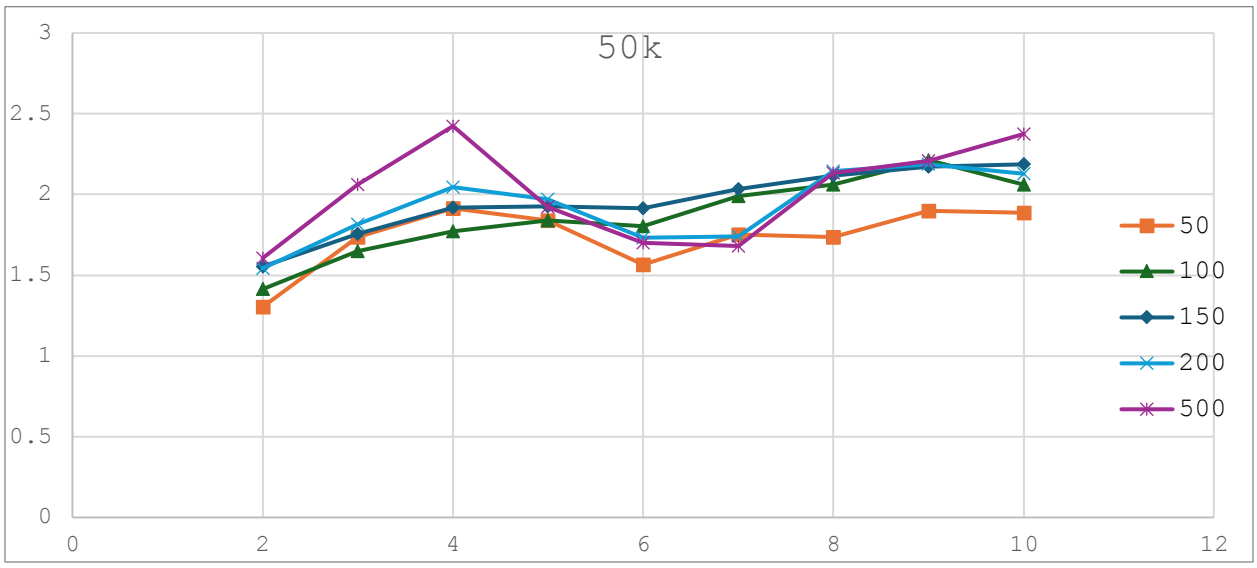
- Population size {10k, 20k, 50k, 100k, 500k}
- Simulation time {50, 100, 150, 200, 500}
- Thread count {2,3,4,5,6,7,8,9,10}

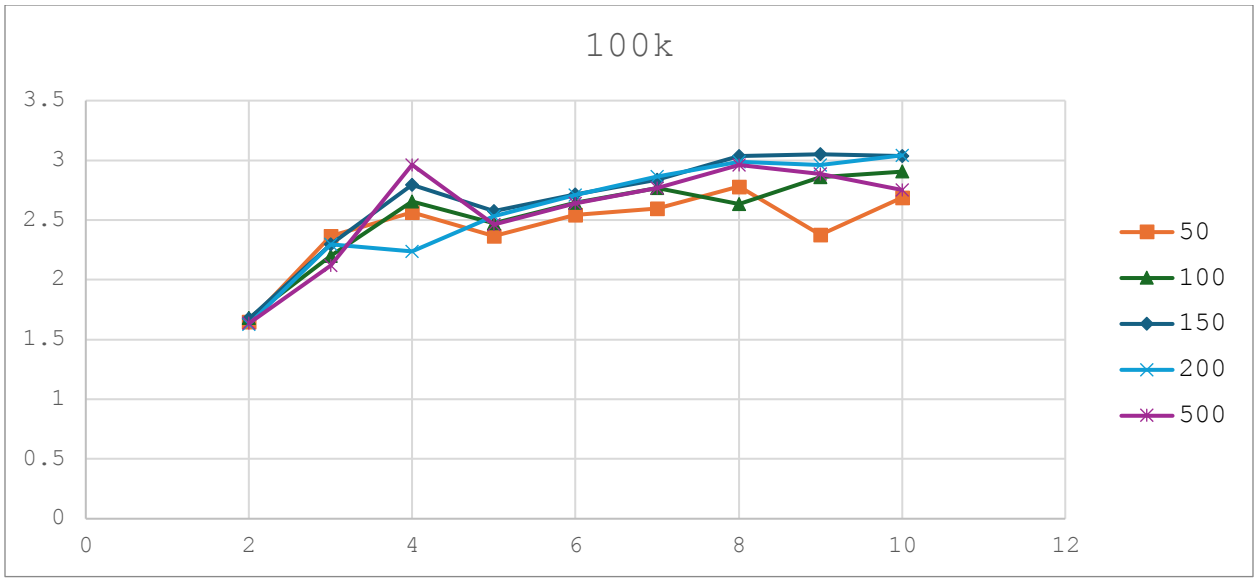
How to read the graphs:

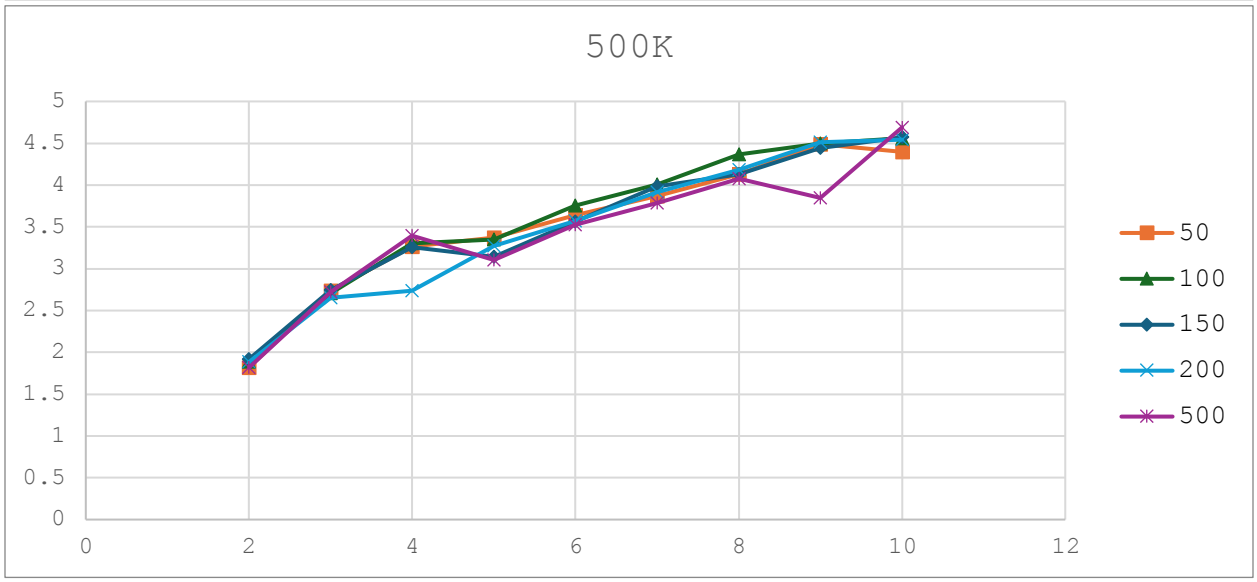
- The graph title represents the population size
- The legend on the right represents the simulation time
- On the X axis it is thread count
- On the Y axis it is speedup


10k
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
12
50
100
150
200
500


20k
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
12
50
100
150
200
500


50k
3
2.5
2
1.5
1
0.5
0
0
2
4
6
8
10
12
50
100
150
200
500





Observations

For 10k population we can observe that due to the parallelization overhead, it is worth it only for longer simulation times and only with 3 or 4 threads, 3 threads seemingly being a sweet spot where only 1 thread is slower than the serial version.

For 20k population we can see that the speed is increased up until 7 threads, where the smallest simulation times begin performing worse than serial. The sweet spot seems again to be around 3 or 4 threads.

For 50k population we, for the first time, have a speedup in all areas. The speedup is roughly the same, being quite stagnant, the sweet spot seems to be around 4 threads, after that the speedup goes down and only comes back around at 8 threads.

For 100k population the speedup steadily grows, once again reaching the sweet spot of 4 threads, before going down and then going up again, plateauing at around 8 threads.

For 500k population the benefits of parallelization are evidently the strongest, the growth rate being steady, and the difference between the short and long simulation time being negligible. The sweet spot, once again, seems to be at 4 threads, and the speedup plateauing at 8 threads.

Final Remarks:

- There seems to be something going on with the 4 threads situation, always being a spike in performance there followed by a valley that that will that plateaus at 8 threads
- As expected, the speedup increases with the workload, be it in the form of a longer running simulation, or in the form of a larger volume of the data, since the parallelization overhead gets relatively smaller
- contagionZone isn't a critical area in the current implementation. Even though every thread writes to it without a mutext, because writing is an atomic operation and they write the same thing, the result is that they can't ruin the data integrity
- Padding for contagionZone: since contagionZone is randomly accessed by any thread I feared that false sharing will appear, thus the padding. But it didn't really make a difference, the measurements were inconclusive, sometimes one variant being slower sometimes the other.
- Writing overhead: while padding had no tangible effect on speedup, what did have was writing only if the contagion zone wasn't set to 1 already. Thus, indicating that the overhead for bringing a variable in memory and comparing it to 0 is small enough when compared to writing to be worth checking every time.