**APD Project 2: Epidemics**

Brebu Iasmin-Marian, 3C.1.1

**Overview**

In parallel V2, 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.

The parallel V1 version is the same too, but instead of each thread only dealing with one chunk for the duration of its life, we use pragma parallel for, meaning threads are assigned multiple smaller chunks.

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.

**Parallel V1: pragma parallel for**

Outside the loop that counts the simulation time, create the threads which will have work assigned. This loop cannot be parallelized.

Inside the loop there are 2 separate loops that correspond to the main phases of the algorithm:

* First loop iterates over every person and updates their positions
* Second loop updates the status of every person

There is no need for a barrier to separate these two phases because of the implicit synchronization after pragma for.

After that we reset the contagionZone and put a barrier to make sure the next round doesn’t start before we are done resetting.

**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).

For running without caring about thread count (useful for the parallel V1) there is a runNoThread.sh. Same as before, need to modify the print statements.

A cleaned-up results batch together with graphs is in apdGraphsAssignment2.xlsx. Here there are the measurements for V1 as well as V2, and the comparisons for their best cases.

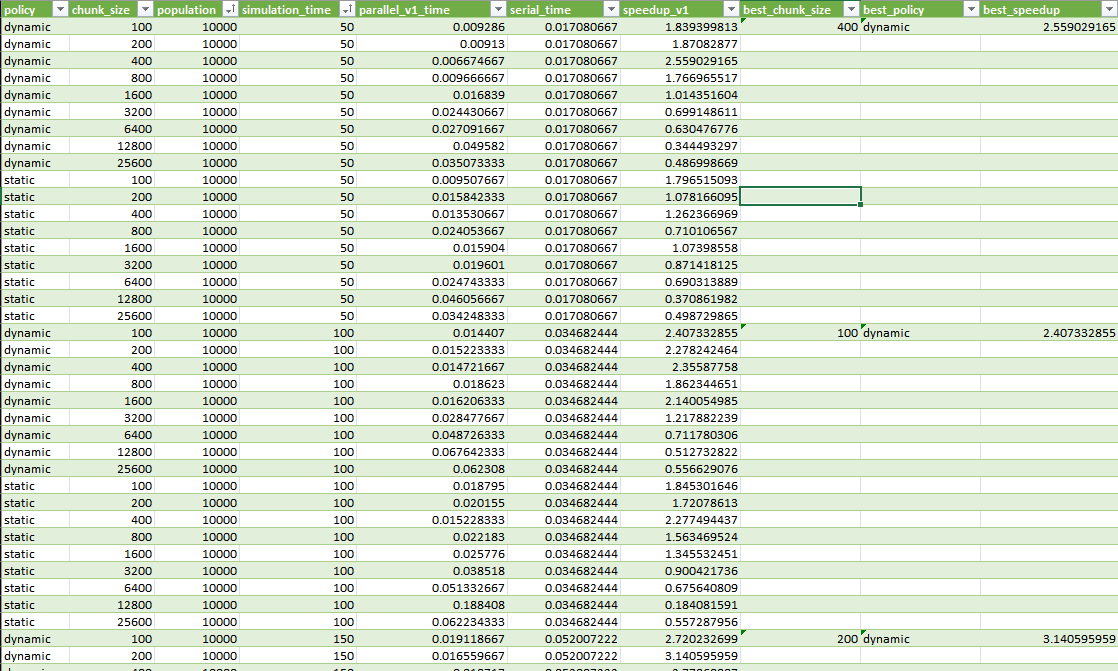
**Measurements results for parallel V1**

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}
* Scheduling policies {dynamic, static}
* Chunk sizes {100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600}

The results are in the excel file in the parallel\_v1 sheet.

Reading the table is straightforward. Serial\_time is the same for each population/simulation\_time pairing because it is the average from the measurements from last assignment. Best\_chunk\_size, best\_policy are the best policy and chunk size for the population/simulation\_time pairing, with best\_speedup being the resulting speedup.



**Measurements results for parallel V2**

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}

The results are in the excel file in the parallel\_v2 sheet.

Best\_thread\_count represents the thread count which got the best speedup for the populationSize/simulationTime pairing.

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

**Comparison**

In the excel file in the comparison sheet I have compared the best cases for each parallel version.

Reading the graph:

* The title represents the population size
* The bottom series represents the simulation times
* V1 is blue, V2 is orange
* On the y axis there is the speedup

We can observe that V1 consistently outperforms V2 in the best case scenarios. The difference beeing especially pronounced in on smaller workloads.

This doesn’t necessarily mean that V1 is better, because we can’t automatically assign how many threads, what chunk size and what policy to use during runtime. In V2, based on these measurements, we could have an if at the start to choose the best parameters for the workload given. Or to simply skip parallelization entierly.

**Observations for V1**

The dynamic mode consistently outperforms the static mode in most situations. Most likely because the workload is unbalanced. Being quite unpredictable when you must write to contagionZone, when you have to increase/decrease counters, and so on.

Best chunk size for the pairings were quite unexpected, with there being a trend for smaller workloads doing better with bigger chunk sizes, and for best chunk sizes for bigger workloads being quite unpredictable, sometimes the smallest sizes outperforming the bigger ones.

**Observations for V2**

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 mutex, 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.