APD Project 3: Epidemics

Brebu Iasmin-Marian, 3C.1.1

Overview

The population is distributed equally among each task. The excess is evenly distributed across the first tasks. For each iteration a task:

- Updates the positions of its population
- If a person is infected, then it marks the position as being infected in a local contagion zone
- [**synchronization**] applies MPI_Reduceall with the MPI_MAX operation to combine all local contagion zones into the true contagion zone
- Determines the status of each person based on its current status and if it is in a contagious zone
- Cleans the contagion zone

The serial version is conceptually the same, but as it takes care of the entire population it doesn't need the reduce call to determine the true contagion zone.

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. All we need to do is comment out the speedup and verification printfs and add a printf in a desired format (csv).

A cleaned-up results batch together with graphs is in apdGraphsAssignment3.xlsx.

Measurements results

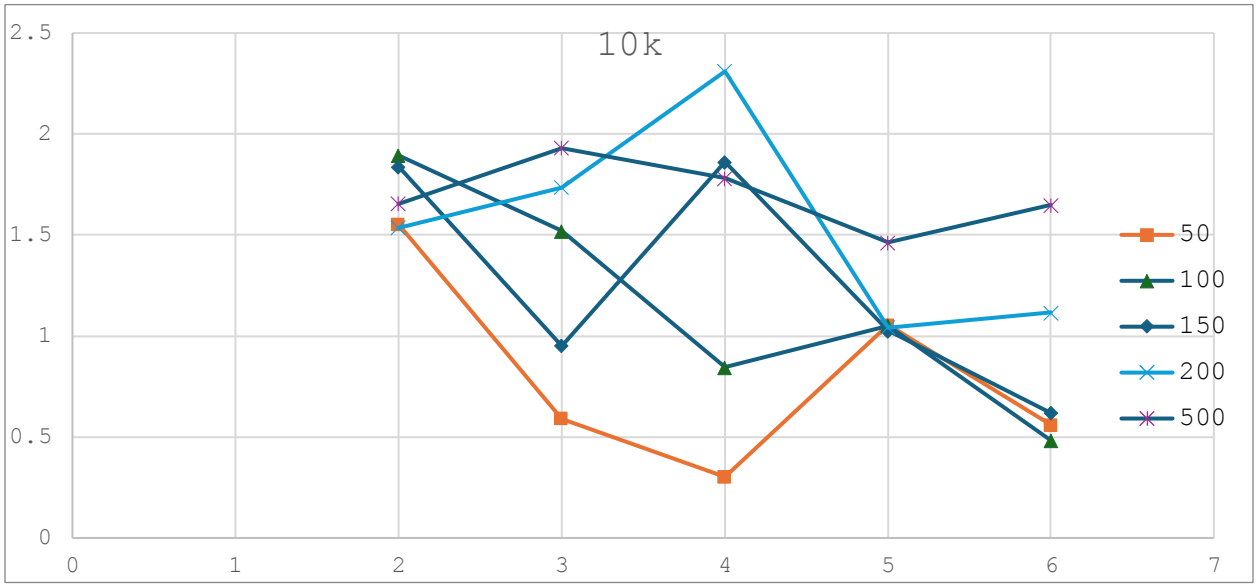
The measurements have been done doing all possible combinations of the following parameters:

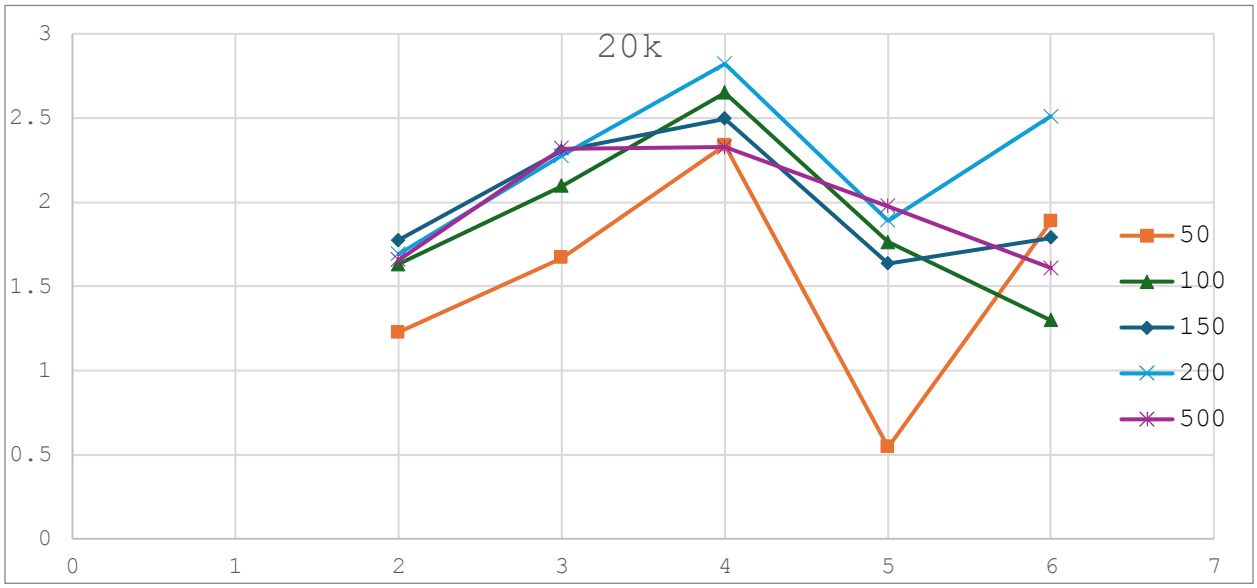
- Population size {10k, 20k, 50k, 100k, 500k}
- Simulation time {50, 100, 150, 200, 500}
- Thread count {2,3,4,5,6}

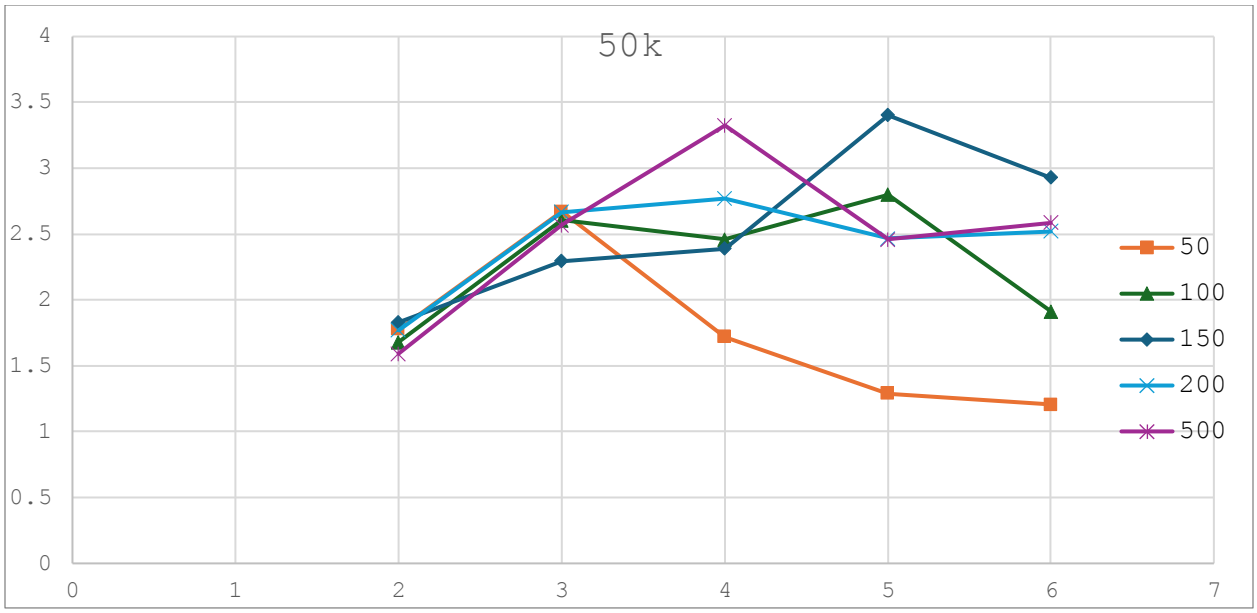
How to read the graphs:

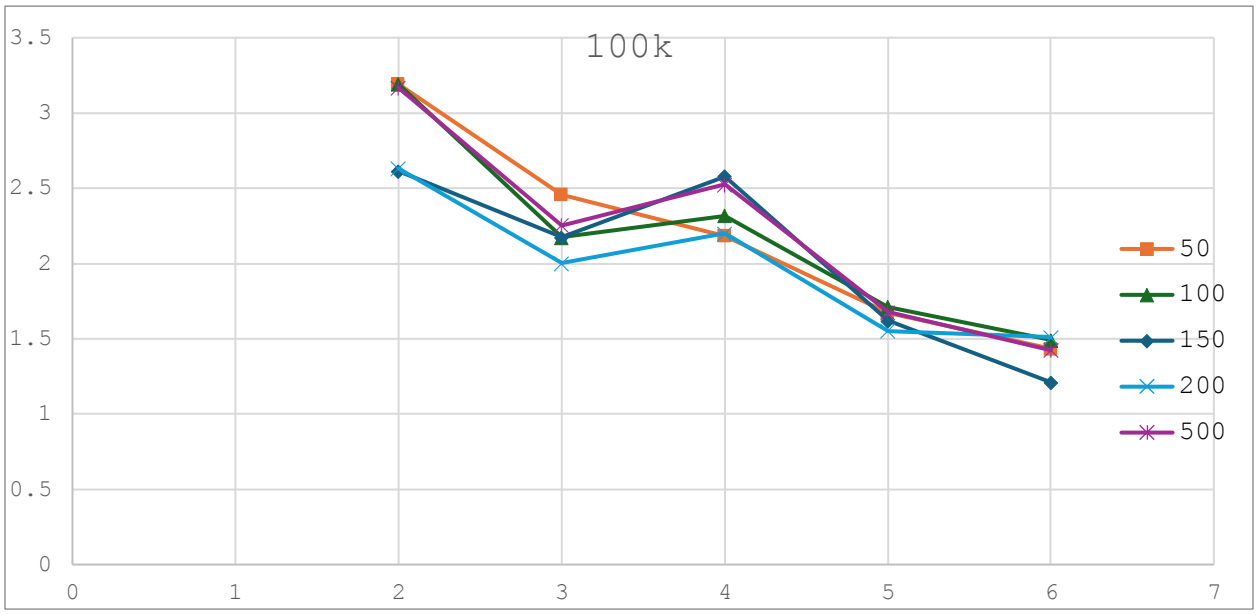
- The graph title represents the population size
- The legend on the right represents the simulation time
- On the X axis it is task count
- On the Y axis it is speedup

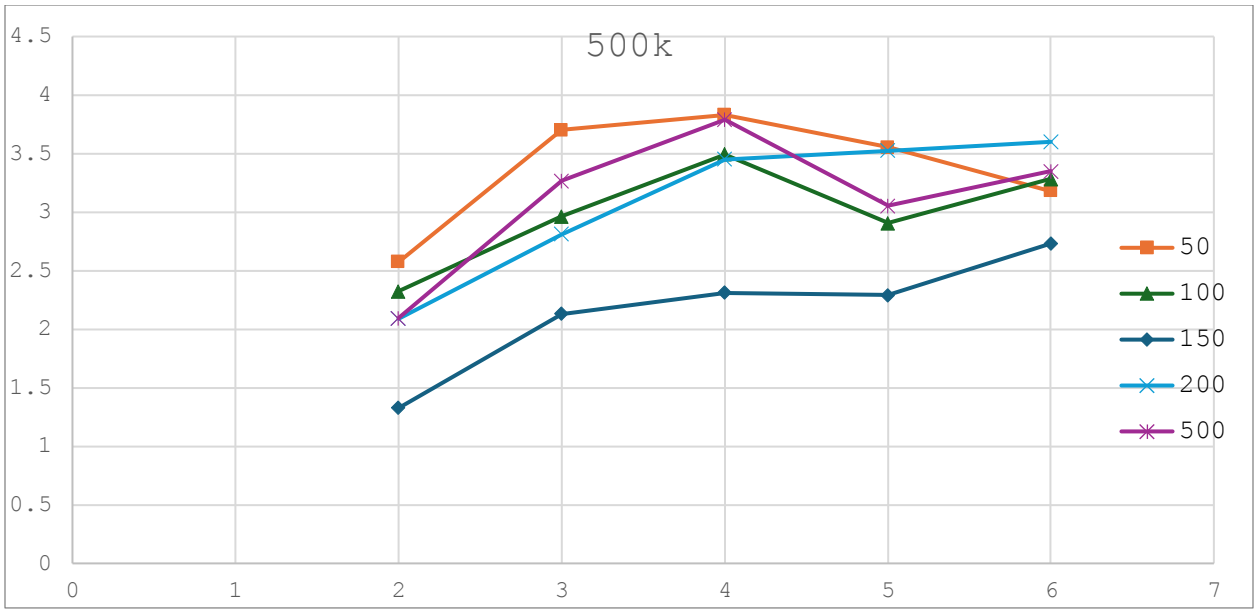
Graphs for speedup compared to serial version:



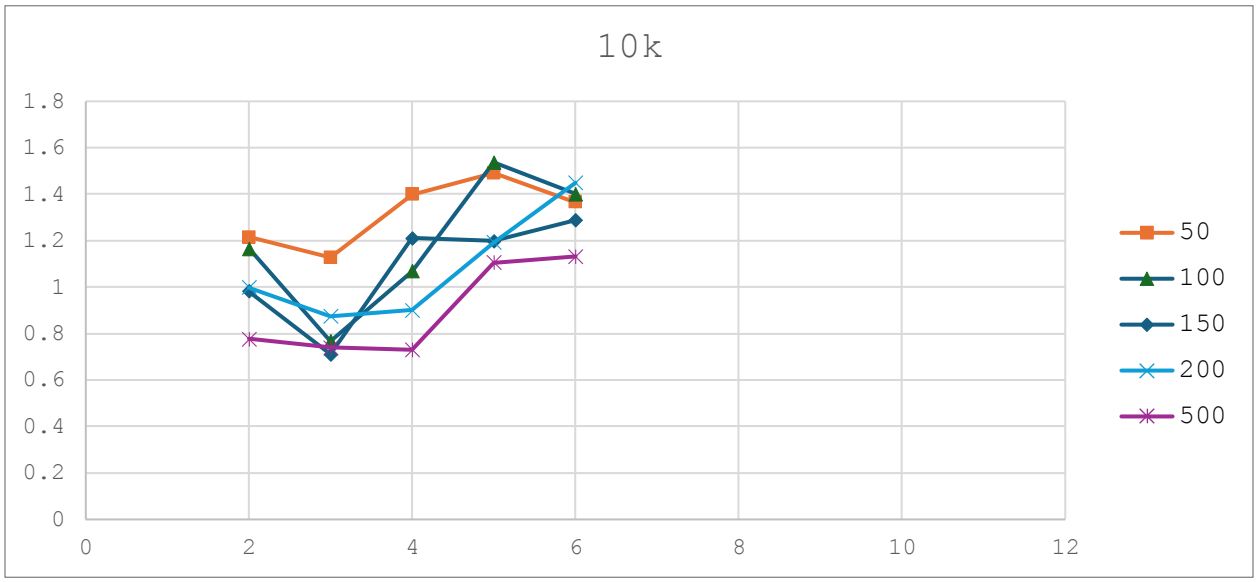


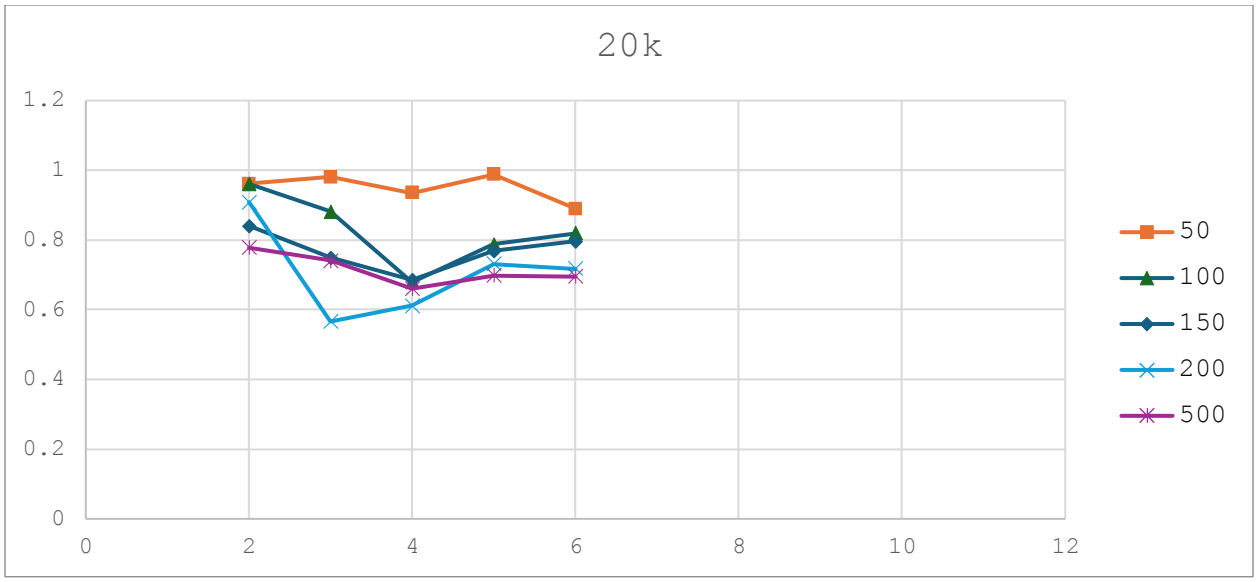

50k
4
3.5
3
2.5
2
1.5
1
0.5
0
0
1
2
3
4
5
6
7
50
100
150
200
500

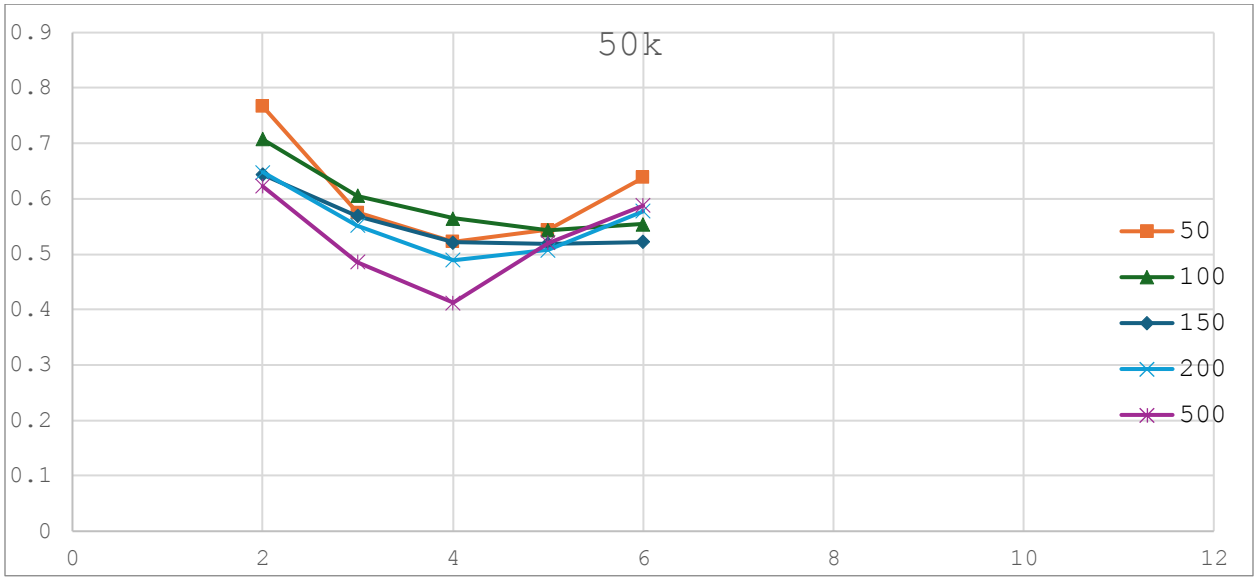

100k
3.5
3
2.5
2
1.5
1
0.5
0
0
1
2
3
4
5
6
7
50
100
150
200
500

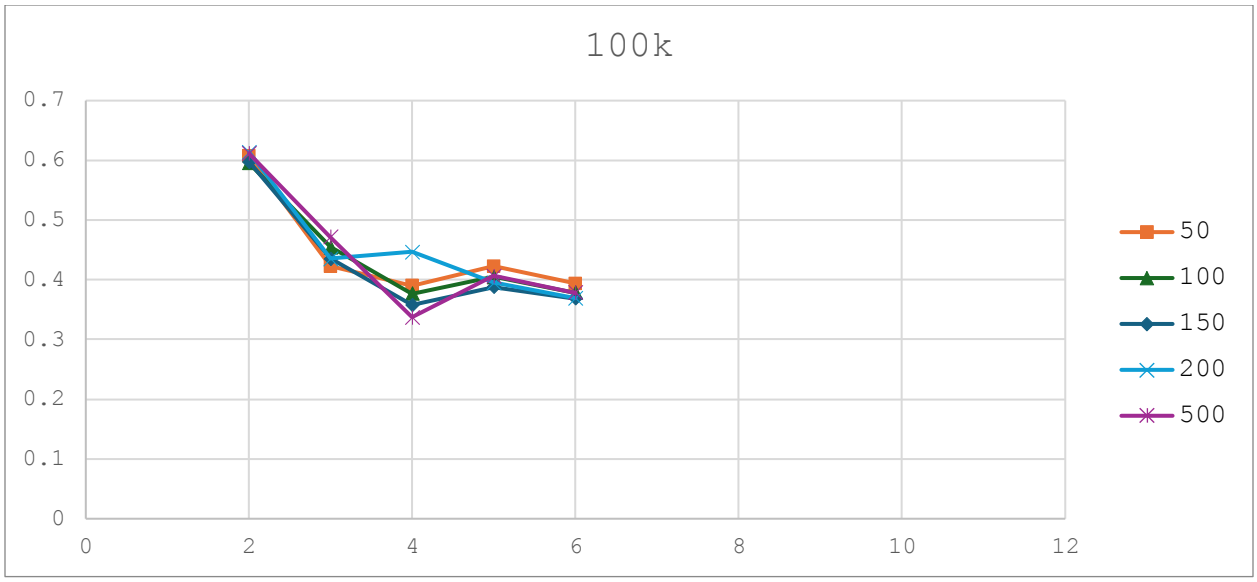


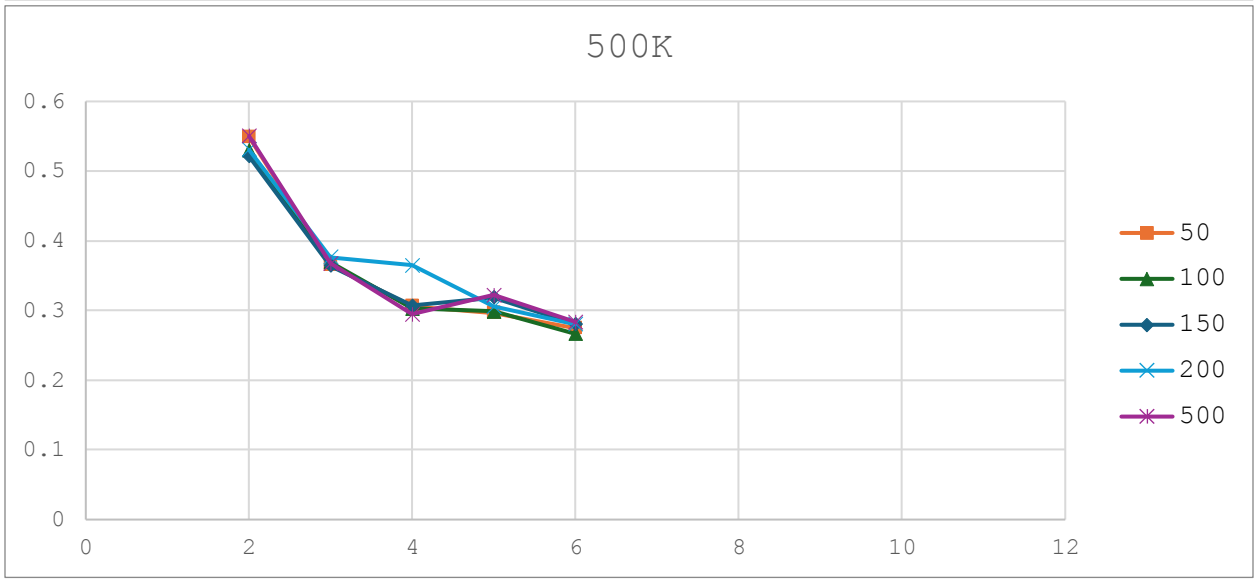
Graphs for speedup compared to a parallel version what uses the same thread count as there are tasks.




20k
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
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
50
100
150
200
500





Observations

- MPI is more useful for smaller amounts of transferred data. Though this might be because I am transferring the person_t struct using MPI_BYTE as a workaround.
- The spike in performance at 4 tasks (because it's run on the same machine, so actually 4 threads) is still apparent, though not as pronounced.

Final Remarks:

- Why is MPI better for smaller data sizes? There are 3 notable communications being done:
    - Scattering the entire population to the local personsMPILocal of each task. Notable because it's load increases with population size.

    - Reducing all contagion zones to obtain the true contagion zone. Notable because it's done at every iteration
    - Gathering the personsMPILocal of all tasks into the master task for writing and comparing with the serial result. Notable because it's load increases with population size.
- I believe is the first and last point that results in such a bad performance. Mainly because we can clearly see when compared to a parallel implementation that at higher population it starts to perform worse and worse. If it was because of the second point then we would see a much worse performance at lower populations too, but at 10k population it begins to outperform the parallel version starting with 4 tasks/threads.