Infection Analysis - MPI

Bruno Morelli Cristian Lo Muto Vincenzo Martelli

Contents

0.1 Introduction

Scientists increasingly use computer simulations to study complex phenomena. In this project is implemented a program that simulates how a virus spreads over time in a population of individuals. The program considers N individuals that move in a rectangular area with linear motion and velocity v (each individual following a different direction). Some individuals are initially infected. If an individual remains close to (at least one) infected individual for more than 10 minutes, it becomes infected. After 10 days, an infected individual recovers and becomes immune. Immune individuals do not become infected and do not infect others. An immune individual becomes susceptible again (i.e., it can be infected) after 3 months. The overall area is split into smaller rectangular sub-areas representing countries. The program outputs, at the end of each simulated day, the overall number of susceptible, infected, and immune individuals in each country. An individual belongs to a country if it is in that country at the end of the day.

0.2 Approach and Assumptions

The technology that has been used is MPI since it is very suitable to run scientific simulations on distributed systems because it allows the developer full control on the program execution.

To model the movement of the persons an x-velocity and a y-velocity and a direction that is either positive or negative have been assigned to each one. When a person arrives to the border of the grid its direction is inverted. The velocity is initially set to a random value between '-v' and 'v' where 'v' is a parameter. The world has been modeled as a grid, so each person is placed on a cell. Furthermore, the grid is divided in macro blocks corresponding to the countries. Countries are blocks of equal size that perfectly fill the world, for simplicity is assumed that the user will use valid arguments, that is the parameter 'W' is a multiple of the parameter 'w' and 'L' is a multiple of 'l'. In order to improve the quality of the simulation the people that are initially infected start with an infection-timer that is a random value between zero and ten days and the velocity of the persons change each day. In order to improve MPI performances the persons have been split equally to each process, if the number of persons cannot be divided by the number of processes than a number of 'ghost' persons are added until they are a multiple of the processes. These 'ghost' are placed outside the map (coordinates -1,-1) and do not contribute to the infection computation.

0.3 Design

This is the base structure to represent a person:

```c
typedef struct person{
    int id;
    int x,y;
    float vx,vy;
    InfectionStatus status;
    int time;
} Person;
```

- 'id': used to check if the person real or is a 'ghost'
- the coordinates x and y: integers because the position on the grid must be integer
- the velocity: float to better compute the movement
- the status: susceptible if the person can be infected, infected, immune if three months have not yet passed since the last time the person was infected, in_contact if the person has been in contact with an infected person for less than ten minutes

```c
typedef enum infectionStatus{
    SUSCEPTIBLE, INFECTED, IMMUNE, IN_CONTACT
} InfectionStatus;
```

- time: how much time has passed since the person entered this status.

Each process at the start of the program generates an equal number of persons and a local copy of the grid. The grid is a matrix with Boolean values, the size is W*L and in each position the value is 1 if there is an infected person in that location and 0 if not.

0.4 Algorithm

At each timestep each process iterates over all the people and perform these local steps:

- move
- increase persons internal time and check if an infected person should become immune or if an immune person should become susceptible.

Then the steps to perform the new infection are performed. The algorithm is easy, each process fills an array with the positions of its infected and sends the number to the process rank 0. The process 0 then performs a gatherv and populates an array with all the infected positions of all the process than is then broadcasted to every process. At this point the processes put the cell of the grid corresponding to the infected persons to 1.

Finally, the processes iterate over their persons and for each one they check if the square of side 'd' centered in the person location there is an infected person. The complexity is therefore O(d*d*N_proc). If d*d is bigger than the number of infected persons than the program automatically decides to simply iterates over the infected list instead of placing the infected positions on the grid so the final complexity is O(min(d*d*N_proc, N_inf*N_proc)).
Where N_proc is the number of people managed by each process and N_inf is the number of infected people at each timestep.
After this all the cells previously set to 1 are brought back to 0.

0.5 Test

0.5.1 Installation

1. Install MPI
2. Set up MPI: move in the directory where MPI was installed and run the following commands:
   (a) export TMPDIR=/tmp
   (b) export PATH="$HOME/opt/usr/local/bin$:PATH"
3. Compile:
   (a) mpicc -o virus mpi_proj.c utils
4. Run:
   (a) mpirun -np PROCESSES ./virus N I W L w l v d t

Substituting in place of the parameters:

- PROCESSES: number of processes desired
- N: number of persons
- I: number of persons that are initially infected
- W: width of the grid
- L: length of the grid
- w: width of the single country
- l: length of the single country
- v: maximum velocity of the persons
- d: maximum distance of infection
- t: simulation timestep

0.5.2 Performances

In the tables below are reported twenty configurations used to test the performance of the program, these differs in the number of cores used, number of people, number of countries and infection distance. The execution time reported is relative to a simulation of one day.
The program has been tested in a distributed using two ubuntu virtual machines connected over a bridge network but the numbers reported in the tables below are the result of a local test run on a MacBook Pro with M2pro processor.

2 processes

N	l	W	L	w	l	v	d	t	Execution Time (seconds)
39872	10	10000	10000	2000	5000	2	25	100	3.502257
39872	10	10000	10000	2000	5000	2	50	100	5.921627
398720	10	10000	10000	2000	5000	2	5	100	32.569889
398720	10	10000	10000	2000	5000	2	25	100	55.493428
398720	10	10000	10000	5000	5000	2	25	100	40.417795
398720	100	10000	10000	2000	5000	2	5	100	125.781900

10 processes

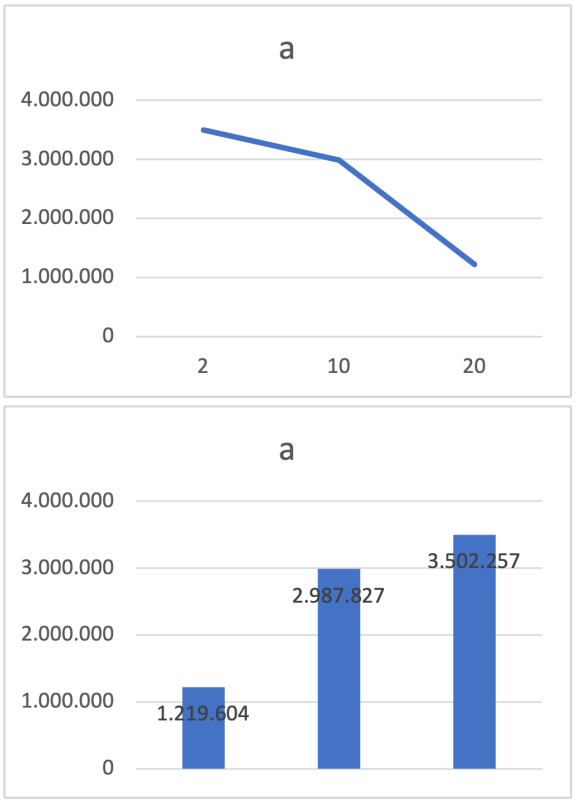
N	l	W	L	w	l	v	d	t	Execution Time (seconds)
39872	10	10000	10000	2000	5000	2	25	100	2.987827
39872	10	10000	10000	2000	5000	2	50	100	4.866452
398720	10	10000	10000	2000	5000	2	5	100	15.998688
398720	10	10000	10000	2000	5000	2	25	100	21.201036
398720	10	10000	10000	5000	5000	2	25	100	16.825681
398720	100	10000	10000	2000	5000	2	5	100	40.649228
398720	100	10000	10000	2000	5000	2	50	100	232.706027

20 processes

N	l	W	L	w	l	v	d	t	Execution Time (seconds)
39872	10	10000	10000	2000	5000	2	25	100	1.219604
39872	10	10000	10000	2000	5000	2	50	100	1.780011
398720	10	10000	10000	2000	5000	2	5	100	6.203934
398720	10	10000	10000	2000	5000	2	25	100	10.780575
398720	10	10000	10000	5000	5000	2	25	100	8.368974
398720	100	10000	10000	2000	5000	2	5	100	18.007809
398720	100	10000	10000	2000	5000	2	50	100	114.404981

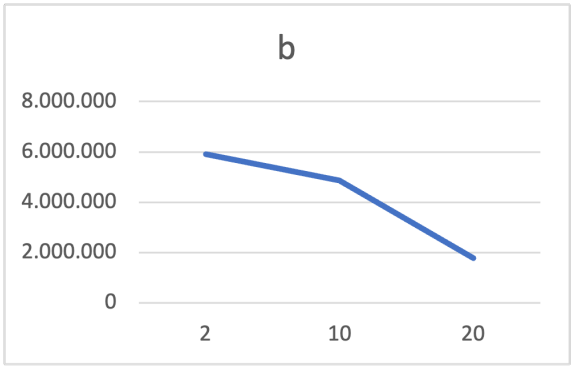
In the following bar diagrams it is shown how the different processes configurations performs against each other on the previous tests ('a' is the first, 'b' the second and so on), the first column refers to the 20 processes run, the second to 10 and the last to 2.

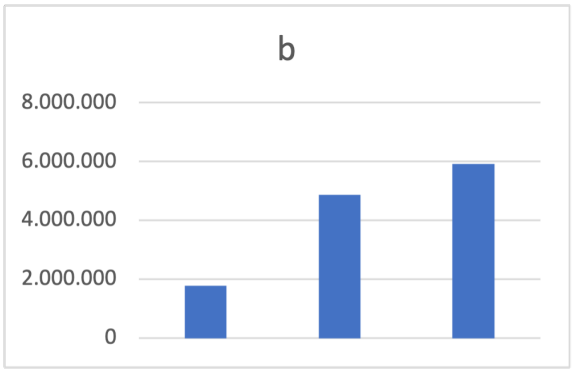
0.5.3 Test a



The easiest test of all, the 20 processes configuration is faster than the others but is only one three times faster than the 2 processes. There is not much difference between 10 and 2 processes.

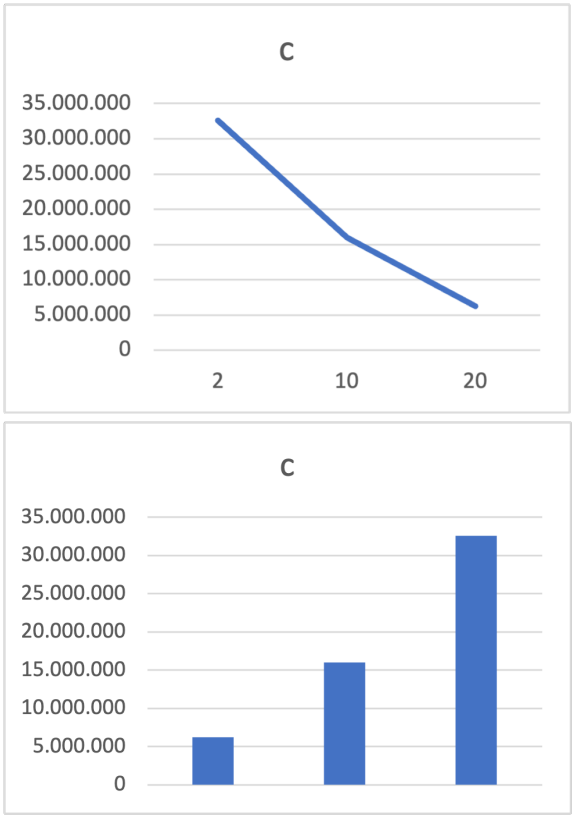
0.5.4 Test b





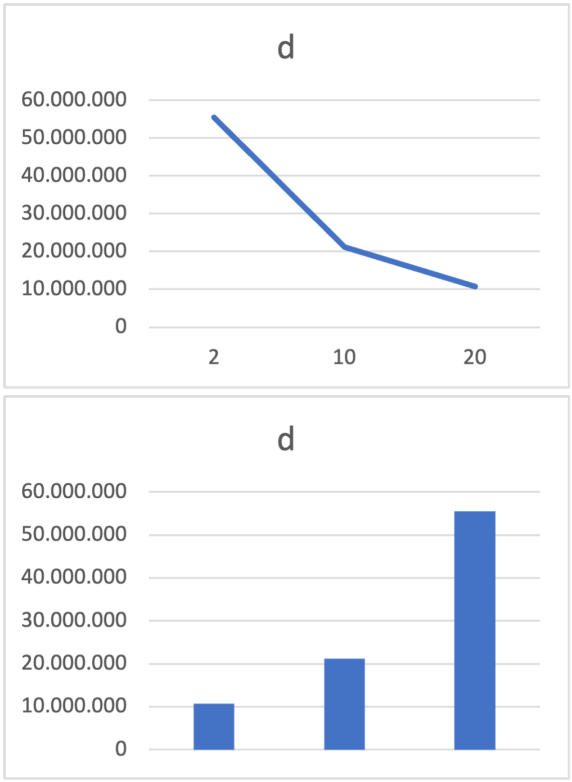
Similar consideration as test ‘a’

0.5.5 Test c



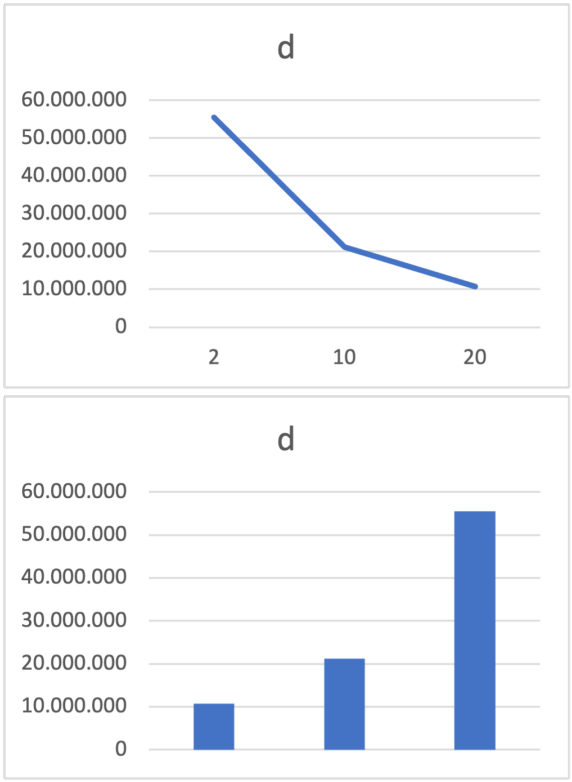
Here the difference between the various configurations is bigger, the 20 processes configuration is more than double faster than the 2 processes with respect to previous tests. Finally, there is a clear difference between 10 and 2 processes that was not clear in the other tests. This is due to the fact the number of people N is 10 times bigger than the one used in test 'a' and 'b', 398720 versus 39872.

0.5.6 Test d



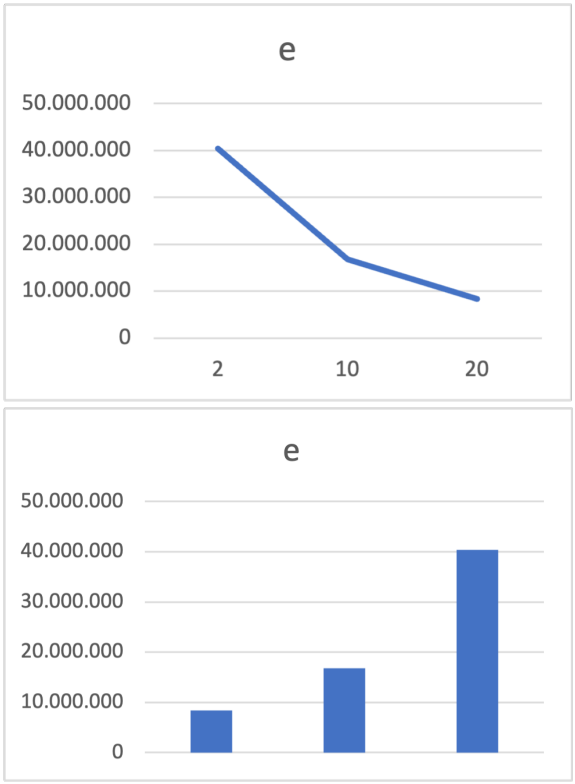
Similar consideration of test ‘c’ but with an even bigger difference from 10 processes to 2.

0.5.7 Test e



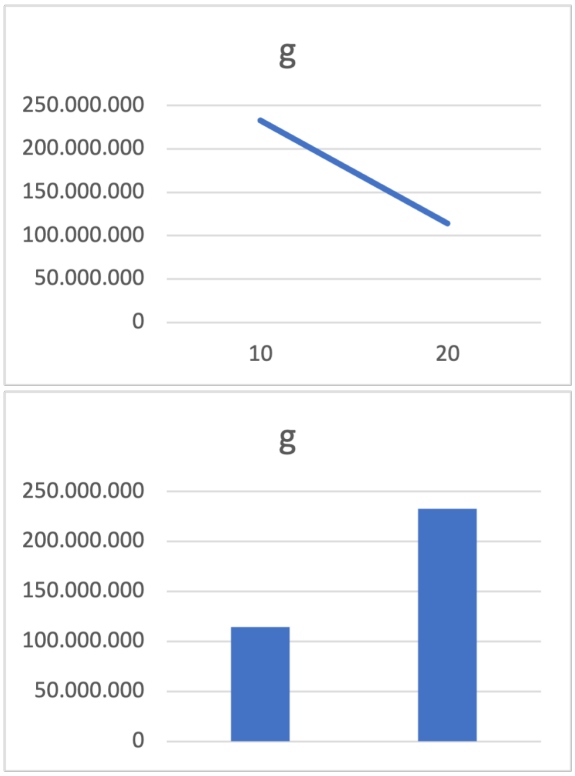
The difference between the various configuration is similar to the one of the previous two test. All the processes run faster than in test 'd' this is due to the smaller number of countries of this test.

0.5.8 Test f

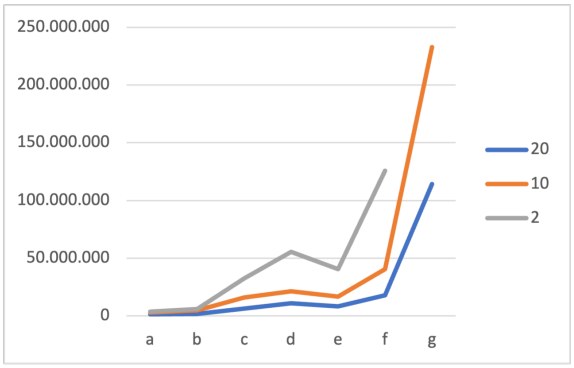


In this test all the processes run slower (with roughly the same proportion). This is the effect of more infected people, so the number of data that has to be allocated and transmitted is bigger.

0.5.9 Test g



This test produces slower performances from both 10 and 20 configurations due to the high infection distance that is an essential parameter to play around to measure the performance of this algorithm. The 50 meters used here is clearly a far bigger 'd' than the one really needed, as an example the recommended safety distance for covid-19 was 3 meters.
Here there is an overview of how the three configurations hold up against each other in the previous tests.



From this chart is clear that more difficult tests result in larger performance gaps.

0.6 Challenges

The main source of problems can come from the limit of the size of the allocable matrix, based on test the limit size was estimated as $10\hat{9}$ bite (125MB), the algorithm would recognize values bigger than this and use the version without a grid. Good results have been obtained even with grid value around $10\hat{}14m^2$ (earth size is 5,101 x $10^{14}m^2$). As an example, the following test was executed in 5.011255 seconds.

39872	100	10000000	10000000	200000	500000	2	25	100

The bottleneck, for the implementation of the grid-less, is the number of the infected people. In this case the simulation granularity can be increased, the former example with 10000 infected people runs in half a second with a granularity of 3600 seconds that is the equivalent of an hour.

0.7 Conclusions

The program can run with various configurations of parameters, offers its best performances when the grid can be instantiated and can run the algorithm with time depending on the distance since this parameter in real sets will be small.
From the results of the test the algorithm does not results to be embarrassing parallel since the gained speedup when passing from 2 to 20 processes is around 6 instead of the theoretical speedup of 10, even though the speedup from 10 to 20 processes is often bigger than the theoretical speedup of 2 reaching sometimes 3. The fact that the practical speedup in this last case is bigger than the expected can be explained by the noise that empirical tests bring. Since the speedup increases the more the size of the test increases it can be deducted that it is hold back by a fixed part of the program that cannot be parallelized while the parallel part is rightfully parallelized.