A Simple Model for Virus Spreading

Middleware Technologies for Distributed Systems

Chiara Marzano
Massimiliano Nigro
Daniele Paletti

29/09/2021

Contents

1 Overview

This project implements a simplified virus spreading model built using MPI. The model simulates the movement of individuals given a certain speed, a starting position and direction in a world of rectangular countries. When two individuals are close for 10 or more minutes and one of them is infected, the other becomes infected too. An individual is infected for 10 days and, once they become healthy, they stay immune for 3 months.

2 Technology choices

Following the specifications of this project, we decided to develop this project using **MPI**, specifically the **OpenMPI** implementation. The decision to use MPI against Apache Spark is based mainly on the following reasons:

- simulations usually fall the category of **compute-intensive** application, as its resources are mainly devoted to computations, and this project can be considered as such, therefore justifying our choice. On the other hand, Apache Spark is more suited for data-intensive applications and most of its functionalities have little to no application in our specific case.
- MPI acts on a lower level and provides fine-grain control over the execution and parallelization of tasks, which allowed us to test different approaches and choose the most suitable.

Specifically, we chose the Open MPI implementation as it is one of the most used and as it was created merging the best characteristics of previously popular MPI implementations.

3 Assumptions

Besides the assumptions of the project specification, we assumed that, once an individual reaches the border of the world, they relocate to a randomly chosen position. This assumption can be considered mimicking an individual travelling without contact as if in a car or plane ride and, for this reason, we believe it can be considered realistic.

4 Structure

1. Setup of MPI communication
2. Input parsing: Input file is parsed and parameters are stored;
3. Initialization of World: Using the parsed parameters, the world is initialized, countries are placed in the World and each thread creates and places its individuals;

4. Initialization of JSON helper class: A JsonHandler object is created to aid storage, serialization and deserialization of data;

5. Iterations over the day:

   (a) Update positions of individuals: each individual is moved on a straight line based on their initial position, speed and direction

   (b) Collect list of infected: each thread creates a list of its infected individuals and shares it with the others

   (c) Share and merge lists: a global list is created merging the local lists and then it is broadcast;

   (d) Compute newly infected: based on the previously shared list, the infected individuals are computed and the infected list is updated;

6. Compute and update statistics: At the end of the day, each thread computes local statistics. Then they are merged and the global statistics are printed.

5 Design choices

5.1 Configuration file and input parsing

To ease input operations and testing, we decided to use a JSON file as input. To be parsed correctly by our implementation, the file has to be of the form:

```
{
  "individuals_number": NUM_INDIVIDUALS,
  "infected_number": NUM_INFECTED,
  "world": {
    "width": WIDTH_WORLD,
    "length": LENGTH_WORLD
  },
  "countries": [
    {
      "width": WIDTH_COUNTRY,
      "length": LENGTH_COUNTRY
    },
    ...
  ],
  "velocity": VELOCITY,
  "days": DAYS,
  "maximum_spreading_distance": DISTANCE,
  "time_step": TIMESTEP
}
```

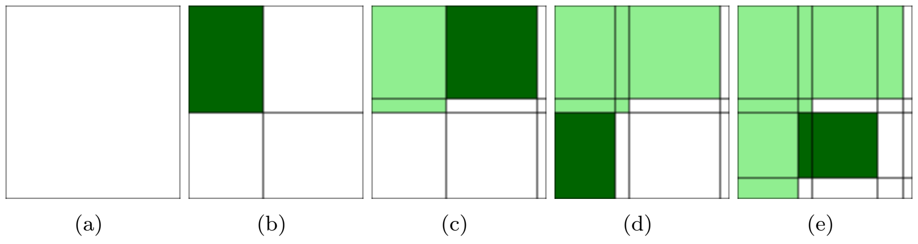


Figure 1: First steps of the algorithm. Light green cells are occupied, dark green cells are the latest placed.

5.2 Country placement

When presented with a *World* and a set of countries to be placed, we selected an algorithm that packs a series of rectangles of varying widths and heights into a single enclosing rectangle, with no overlap [1].
This algorithm is based on the concept of cell, which corresponds to the world at the beginning and is further divided once countries are added. A cell is either free or occupied.
The algorithm starts with an empty world (fig:steps a) corresponding to a free grid and iterates over the countries to place. At the first iteration, it anchors the first country on the leftmost corner and, as a cell can only be in one state, it divides the cell as in figure 1.b.
The following countries (figure 1.c, 1.d, 1.e) are anchored in the leftmost available cell in which they can fit without exceeding the size of the world, if they fit in the world but not in a cell, they can be divided in contiguous free cells.

```
// Pseudocode of country placement

for country_to_place in countries_to_place
    find_free_anchor_point()
    if (country fits)
        if (country fits exactly)
        set_anchor_point(country_to_place, free_cell)

        else if (country only fits length-wise)
                divide_cell_further()
                set_anchor_point(country_to_place,
                    free_cell)

        else if (country only fits width-wise)
                divide_cell_further()
                set_anchor_point(country_to_place,
                    free_cell)

        else
            divide_cell_further()
            set_anchor_point(country_to_place,
                free_cell)
```

If a country cannot be placed because it does not fit, it is removed from the list of countries.

5.3 JSON and JSON Handler

JSON is also used to store and send data among the threads, such as the infected at each timestep and the end-of-the-day statistics. In order to send and receive data, the JSON structures have to be serialized and deserialized: for this purpose, we created a **JSON Handler** that provides both operations and relies on **RapidJSON**, a JSON parser and generator for C++.

5.4 MPI Handler and ring communication

MPI Handler is a helper class created to contain all the MPI operations to simplify communications. Among the others, it contains *split_individuals*, that divides the individuals among the available threads, and *ring*, that implements a ring-like topology in which each thread *i* sends to thread *i + 1* and the last one sends to thread *0*. Every time a thread receives a message, it performs a custom cumulative operation based on the occasion (eg: merging the list of infected received with its own local list of infected) thanks to a function pointer provided as input parameter.

5.5 Work division among threads

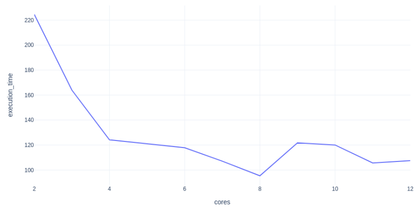
Upon creation of *World*, split_individuals is called and it simply sends every thread the number of individuals it has to place on the *World* and how many of them are infected. All threads receive an equal number of individuals and thread *0* is assigned any residuals.

6 Performance

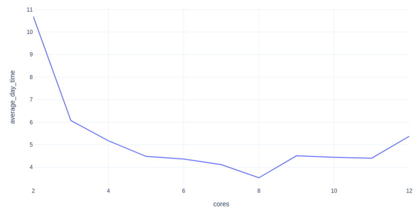
Performances were measured executing on a LAN with a manager and two workers, all three with four cores available, performing 10 runs for each profiling requirement.

6.1 Number of cores

Having 12 total cores available, we tested our application increasing the number of cores at each execution. From the data gathered, we observed the following:



(a)



(b)

Figure 2: cores: (a) Average absolute execution time. (b) Average execution time per day in simulation.

- When moving from 4 to 5 cores, the slope is decreased. This can be explained considering that, up until the previous step, all the cores were available locally and therefore the worker cores went unused. From 5 on, a **network and communication overhead** is introduced and this can also be observed when moving from 8 to 9 cores.
- From 9 to 12, the observed speedup is not high as the chosen simulation could not benefit from added resources. As a matter of fact, as we divide work among the cores by assigning each a portion of the individuals, the number of individuals is too low to require additional resources compared to the added communication overhead of a third worker.

6.2 Number of individuals

We increased the number of individuals gradually from 0 to 11000 and measured the execution time per day. Observing a high variation of the average daily execution time due to intrinsically random components such as movement and placement of individuals, we decided to compute its moving average to perform a more accurate measurement. In the obtained graph, we could observe a generally rising trend in execution time as the number of individuals increases.

6.3 Number of countries

We increased the number of countries gradually and were able to observe that the execution time stays constant. This can be explained by noticing that the number of country does not influence the size of the world in our implementation and most of our complexity comes from iterations on the individuals, therefore the performances do not vary.

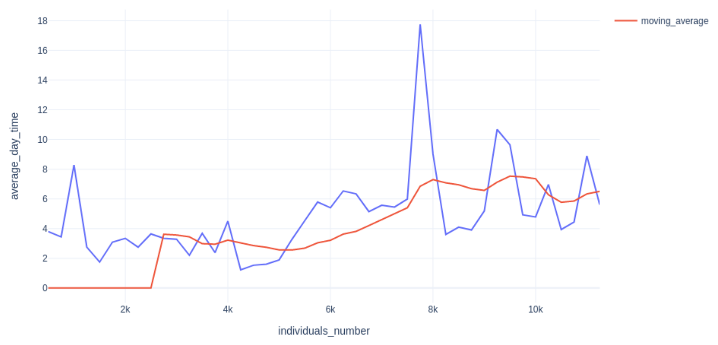
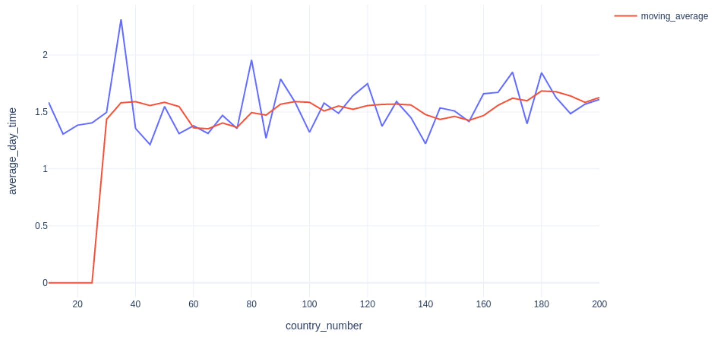


Figure 3: individuals: Average execution time per day in simulation (blue) and related moving average (red).



(a)

Figure 4: countries: Average execution time per day.

References

[1] Matt Perdeck. *Fast Optimizing Rectangle Packing Algorithm for Building CSS Sprites*. en-US. June 2011. URL: `https://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu` (visited on 09/25/2021).