Modeling the propagation of infections in a population

HPC project, Politecnico di Torino

Federico Muscarà
s329756@studenti.polito.it

Marco Parentin
s328112@studenti.polito.it

Leonardo Straccali
s326766@studenti.polito.it

Francesco Zanasi
s333023@studenti.polito.it

***Abstract*—This project implements a parallel algorithm for modeling infection propagation within a population. It leverages High Performance Computing (HPC) techniques. The simulation captures disease spread, agent movement, and dynamic state updates over time. While the core implementations are developed in C (including serial, OpenMP, and CUDA versions), a comprehensive data analysis is performed using Python to evaluate the performance acceleration achieved by the parallel algorithms compared to the serial implementation.**

I. Introduction

THE Kermack–McKendrick model [1], commonly known as the Susceptible-Infected-Recovered (SIR) model, is a well-established mathematical framework for describing the dynamics of infectious disease transmission. In this scheme, individuals transition between compartments based on their interactions. The disease spreads exclusively from an infectious individual to a susceptible one, provided they are within a defined *infection radius*. The probability of a susceptible person becoming infected upon encountering an infectious individual is determined by the disease's *contagious factor* and the susceptible person's *susceptibility*. An incubation period is incorporated, during which an infected person is infectious. Upon the conclusion of this period, the infected individual either recovers or dies. Recovered individuals then have a chance of becoming immune.

We will use the following notation:

- NP: Total number of persons in the model.
- INFP: Initial percentage of infected persons.
- IMM: Initial percentage of immune persons.
- S_AVG: Real number from 0 to 1, representing the susceptibility average of the full population.
- WxH: Area of the region of study in square meters. It is assumed that a person normally occupies a 1m x 1m area, and more than one person can occupy a 1x1 square concurrently.
- ND: Number of simulation days.
- INCUBATION_DAYS: Incubation period in days.
- β: Contagiousness factor of the disease.
- ITH: Infection threshold.
- IRD: Integer Infection radius of the disease, defined as an integer value in meters.
- μ: Probability to recover after being infected.

Instead of solving the partial differential-equations of the SIR model, a useful method for analyzing how the disease evolution changes by varying the different parameters is relying on simulations. However, traditional sequential simulations can be computationally intensive, especially for large numbers of people and extended periods. Therefore, this project explores the development of a parallel algorithm to efficiently simulate these propagation patterns.

The report is organized as follows: Section II presents the project's general structure. Section III describes the fundamental logical analysis common to all simulation model versions. Sections IV through VII detail the specific implementation of the serial, OpenMP, and CUDA algorithms, and the tools used to test and visualize the simulation. Finally, Section IX provides the performance analysis, Section X presents the results of simulations with varying logical parameters and Section XI concludes the report.

II. Setup

In this section the technological choices, application tools, and code organization adopted for the development and simulation of the project are described.

A. Project Organization

The entire project is structured according to a directory hierarchy to separate the various components (source code, scripts, documentation, tests). The most relevant directory for this report is `src`, which is organized as follows:

`src/report`
Directory containing the binary files saved by the different implementations during debug and testing, used for analysis and verification.

`src/config.h`
Global configuration file containing macros and constants (domain dimensions, epidemiological parameters, etc).

`src/structures`
Directory containing custom data structure used by the different implementations.

`src/script/`
Directory containing Python scripts and code for analyzing or debugging the simulation results saved in the `report` folder.

`src/utils/`
Directory containing utility functions.

B. Project Development Environment

To achieve a balance between computational performance and development efficiency, specific programming languages and libraries were employed. **C** was selected for the various implementations, encompassing serial, OpenMP, and CUDA versions, due to their efficiency in high-performance computing contexts. Complementing this, **Python** was utilized for its capabilities in data analysis and plotting.

For robust code versioning, the distributed system **Git** was adopted in conjunction with the hosting platform **GitHub**. We rigorously applied best practices, including a branching model similar to *Git Flow*, which designated a primary `main` branch for stable releases and a `dev` branch for integrating new features. Furthermore, to ensure readability and consistency in commit messages, the *Conventional Commits* standard[1] was followed, utilizing types such as `feat`, `fix`, `refactor`, and `docs`.

At the same time, to maintain well-documented and easily understandable code, a Doxygen-like documentation system was adopted. Although the full Doxygen toolchain was not integrated due to the project's small size, the same nomenclature and syntax (e.g., `@brief`, `@param`, `@return` tags in comments) were used to facilitate future expansion.

Finally, to run simulations on the HPC cluster, the SLURM resource manager was used. The run scripts for all three implementations are located within the `src` directory.

C. Motivations for Architectural Choices

From the outset, three distinct versions of the same program were considered:

- **Serial Version:** serves as a functional and correctness reference, used to validate the baseline models and performances.
- **OpenMP Version:** enables exploitation of multicore CPU parallelism still relying on the same architecture, facilitating comparison with the serial version and providing a performance improvement on machines with multiple threads.
- **CUDA Version:** considered the most promising technology for an agent-based problem, thanks to the massively parallel nature of GPUs, which can handle a large number of entities with repeated local interactions.

All three implementations follow an identical *general logic* so that their final results remain comparable. The only differences lie in how each step is parallelized and/or optimized.

III. Logical Analysis

In this chapter we present the common simulation logic used by the three versions (serial, OpenMP, and CUDA) of the agent-based disease-spread model.

The simulation model is structured into two primary phases. The first is **Population Initialization**. The second phase is the **Simulation Loop for `ND` Days**.

A. Population Initialization

The initialization phase, executed once at the simulation's start, is dedicated to populating the grid by randomly and uniformly placing agents. Concurrently, it assigns each agent an initial state based on `INFP` and `IMM` percentages, configuring relevant parameters accordingly. To achieve this, logical definitions have been established to determine an agent's state based on these characteristic parameters.

An **Immune** individual is defined by a `susceptibility` of 0. It cannot be infected, regardless of the β contagiousness factor.

An **Infected** individual possesses an `incubation_day` count greater than 0 and is not a "newly infected" agent; these individuals have a `susceptibility` greater than 0 as they may become susceptible again post-incubation.

A **Newly Infected** individual is specifically tagged as such, with the logical choice of assigning an `incubation_day` count equal to -1.

Similarly, when individuals transition from an infected state to recovery within a single simulation day, they are designated as **Newly Recovered**.

A **Deceased** agent is simply identified by coordinates lying outside the map (e.g., x or y being negative).

Finally, a **Susceptible** individual is any agent with a `susceptibility` greater than 0 that does not fit into any of the mentioned categories.

After careful analysis of these roles, we agreed that immune agents behave in a particular way within the simulation. The assignment specification does not impose any limit on the number of agents that may occupy the same cell of the grid simultaneously, but if no constraint were imposed, immune agents would have no impact on the simulation. They could simply be ignored, since they never interact with other agents. Since in the later stages of the simulation, the number of immune individuals can become significant, we decided to make their presence meaningful by imposing a maximum occupancy constraint on each grid cell, denoted as MAX_PCELL. This defines the number of available *slots* in a cell. In this way, immune agents occupy space that might otherwise be taken by a susceptible or infected agent, having an effect on the simulation and rendering it more realistic. Consequently, the initialization phase also ensures that the occupancy constraint of each grid cell is respected when placing agents. With the description above in mind, the flowchart reported in Fig. 1 shows the logic of the first phase of the program.

B. Daily Cycle Logic

Each iteration of the simulation loop implements three distinct logical steps.

1) Disease Spread: In this phase, each infected person actively attempts to transmit the disease to susceptible agents located within its `IRD` (Infection Radius Distance). Operating on a two-dimensional grid, infected individuals scan their Moore neighborhood up to `IRD`. Meanwhile, deceased and immune agents remain inactive in this process (though immune individuals continue to occupy their designated space). For every susceptible agent i found within the neighborhood,

[1] https://www.conventionalcommits.org/

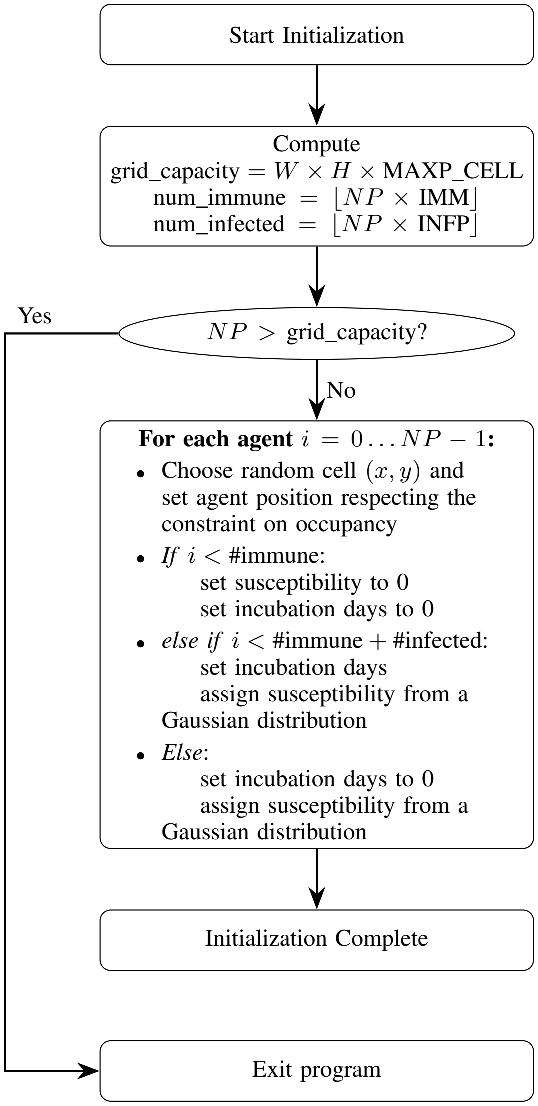

Start Initialization
Compute
grid_capacity = $W \times H \times$ MAXP_CELL
num_immune = $\lfloor NP \times$ IMM$\rfloor$
num_infected = $\lfloor NP \times$ INFP$\rfloor$
Yes
$NP >$ grid_capacity?
No
For each agent $i = 0 \ldots NP - 1$:
Choose random cell (x, y) and set agent position respecting the constraint on occupancy
If $i <$ #immune:
set susceptibility to 0
set incubation days to 0
else if $i <$ #immune + #infected:
set incubation days
assign susceptibility from a Gaussian distribution
Else:
set incubation days to 0
assign susceptibility from a Gaussian distribution
Initialization Complete
Exit program

Fig. 1: Flowchart of the Population Initialization Logic

possessing a susceptibility of S_i, the following condition is checked:

$$S_i \cdot \beta \overset{?}{>} ITH \tag{1}$$

If this condition is met, the agent is immediately labeled as `newly infected`, and the scanning process continues for other susceptible agents in the vicinity.

2) Agent Movement: In this phase, all active agents have the opportunity to move to a neighboring cell in one of eight directions (north, south, east, west, northeast, northwest, southeast, southwest). Because of the maximum-occupancy constraint, the movement step must enforce the appropriate checks and logic to ensure that the amount of persons within each cell never exceeds $MAXP\_CELL$, and must of course prevent invalid moves that would allow an agent to leave the grid. If an agent finds no valid move or attempts an invalid one, it remains in its current position. Therefore, staying still is considered a legitimate "move" in this phase.

3) State Update: In this last phase, for every infected agent that has completed its incubation days, two distinct possibilities arise. The agent may die with a probability of $1 - \mu$; in this scenario, its coordinates are set to a negative value, and the agent ceases to participate in the simulation. Alternatively, the agent may recover with a probability μ, thereby returning to the susceptible state. Subsequent to recovery, there is a 50% chance the agent becomes immune, which sets its susceptibility to 0, and a 50% chance the agent remains susceptible, maintaining its original susceptibility value.

To illustrate the logical architecture described above, the flowchart of Fig. 2 highlights the three main sub-steps within the daily cycle and the state transitions between them.

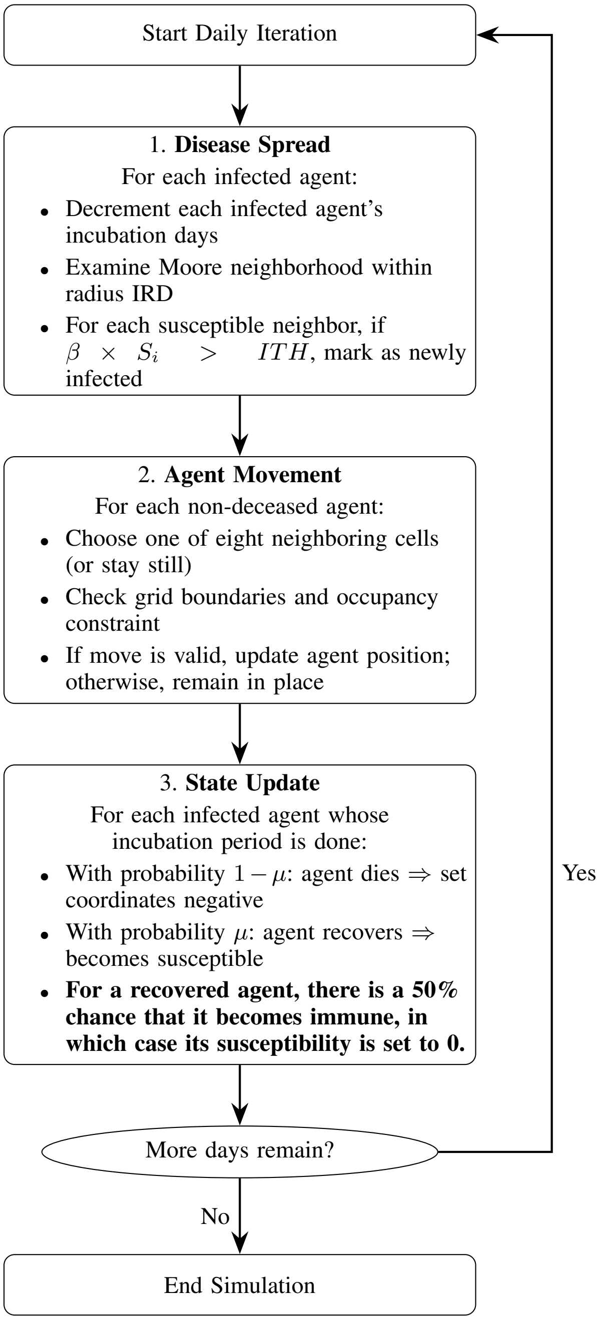

Start Daily Iteration
1. Disease Spread
For each infected agent:
Decrement each infected agent's incubation days
Examine Moore neighborhood within radius IRD
For each susceptible neighbor, if $\beta \times S_i > ITH$, mark as newly infected
2. Agent Movement
For each non-deceased agent:
Choose one of eight neighboring cells (or stay still)
Check grid boundaries and occupancy constraint
If move is valid, update agent position; otherwise, remain in place
3. State Update
For each infected agent whose incubation period is done:
With probability $1 - \mu$: agent dies ⇒ set coordinates negative
With probability μ: agent recovers ⇒ becomes susceptible
For a recovered agent, there is a 50% chance that it becomes immune, in which case its susceptibility is set to 0.
Yes
More days remain?
No
End Simulation

Fig. 2: Flowchart of the Daily Simulation Cycle

IV. Serial Implementation

After analyzing the underlying logic that constitutes our program, this section illustrates the implementation choices made for the serial version. This version serves as a fundamental correctness baseline, enabling validation of the simulation logic in the absence of parallelism. At the same time, it also provides a direct starting point for the OpenMP and CUDA implementations.

The serial version is organized into core functions, each detailed in the next subsections.

The Population initialization phase is managed by the `init_population` function, which is responsible for the random placement of agents on the grid and the assignment of their initial states.

On the other hand, the daily progression of the simulation is encapsulated within the `simulate_one_day` function, implementing the three core logical sub-steps: disease spread, agent movement, and state updates.

Finally, the `main` function is the one that coordinates the overall initialization and orchestrates the simulation loop.

In this context, auxiliary Data Structures (`occupancyMap`, `tupleList`) are used to manage the map of cells and the dynamic list of available coordinates, respectively. Instead, Utilities (`utils.h(.c)`) define essential support functions for managing agent states, generating random numbers, and saving daily reports for debugging purposes.

The code repeatedly uses two global structures:

- `Cell *occupancy_map`: a one-dimensional array representing the $W \times H$ grid, where each element is a `Cell` structure containing an integer `occupancy` (number of agents in that cell) and a pointer to an array of `Person*` of size `MAXP_CELL`.
- `Person **all_persons_pointers`: a global array of pointers to `Person`, used to efficiently store agents in each cell (up to `MAXP_CELL` per cell), thus avoiding dynamic allocations during runtime.

Finally, The key parameters introduced in Section I are defined in `config.h`. In particular, `NP` (the total number of people) is calculated as a percentage of the value `W`×`H`× `MAX_PCELL`.

These constants in `config.h` are parameterized via preprocessor macros, allowing easy modification of the simulator's behavior without changing the code itself. Here is an example:

```
#ifndef W
#define W 3 // Width of the grid
#endif
```

A. *Population Initialization (`init_population`)*

The `init_population(Person *population)` function starts by allocating the necessary memory for the simulation's spatial structures:

```
occupancy_map = malloc(W * H * sizeof(Cell))
all_persons_pointers = malloc(W * H *
    MAXP_CELL * sizeof(Person *));
```

Within these allocations, the `all_persons_pointers` variable is systematically partitioned into contiguous blocks corresponding to each cell. Specifically, the cell defined by coordinates `(x,y)` corresponds, using the flattened index within the data structure , to the person in position position

$$\big((x \cdot H + y) \cdot \texttt{MAXP\_CELL}\big).$$

Following this initial setup, the next step involves creating the list of available coordinates. The `TList` structure is instantiated to store all initially free `(x,y)` pairs, which represent available grid cells. The initialization process then iterates through all `(x,y)` coordinates, adding them to this `available_coords` list and setting the `occupancy` of the corresponding cells to 0. Finally, the selected cell is set to point to `MAXP_CELL` slots of `all_persons_pointers`, properly initialized to `NULL`:

```
TList available_coords = createTList(W * H);
for (int x = 0; x < W; x++) {
    for (int y = 0; y < H; y++) {
        addTuple(available_coords, x, y);
        AT(x,y).occupancy = 0;
        AT(x,y).persons = &
            all_persons_pointers[(xH+y)*
            MAXP_CELL];
        // Initialize all pointers to NULL
        for (int k = 0; k < MAXP_CELL; k++)
            AT(x,y).persons[k] = NULL;
    }
}
```

In this way, instead of each `Cell` having its own separate small array that it has to *malloc* and *free*, it simply points to the beginning of its assigned section within the big `all_persons_pointers` array.

After having correctly initialized the coordinates `TList`, the function proceeds with the random assignment of position and initial state for each of the `NP` agents. For every agent, a random index `idx` is extracted from `available_coords` using `getRandomTupleIndex()`. The corresponding `(x,y)` tuple is then retrieved, and the agent is placed into that specific cell using `addPerson(p, x, y)`. Essentially, when `addPerson(p, x, y)` is called, it directly places the new `Person`'s pointer into the specific pre-allocated memory block within `all_persons_pointers` that belongs to the targeted cell.

During this assignment process, if a cell reaches its maximum capacity (`occupancy == MAXP_CELL`), its corresponding tuple is promptly removed from `available_coords` to prevent any further attempts at assignment to that full cell.

Subsequently, the agent's initial state is determined based on two counters, `num_immune` and `num_infected`, derived from global percentages as `num_immune` $= \lfloor \mathtt{NP} \times \mathtt{IMM} \rfloor$ and `num_infected` $= \lfloor \mathtt{NP} \times \mathtt{INFP} \rfloor$.

Upon completion of all agent placements and state assignments, the `available_coords` list is freed using `freeTList()`, leaving only the fully populated `occupancy_map` in memory.

B. One Day Simulation (`simulate_one_day`)

The `simulate_one_day(Person *population)` function encapsulates all serial operations required to advance the simulation by a single day. Its general structure begins with the preparation of the `newly_changed` array.

This array serves as a temporary buffer to collect individuals undergoing a health status transition within the current simulation day. In other words, it postpones the actual state change for these individuals until the end of the daily simulation: it prevents a *newly infected* agent from immediately transmitting the disease on the same day it contracts it, and similarly, it ensures that a *newly recovered* individual is not re-infected before the start of the next simulation day.

Following this preparation, the function proceeds with a Loop over all agents. Within this primary loop, three logical sub-steps are performed sequentially for each living agent.

1) Disease Spread: During the Disease Spread phase, if an agent `p` is infected, its `incubation_days` counter is immediately decremented:

```
p->incubation_days--;
```

Subsequently, the infected agent scans its surroundings by iterating over a submatrix centered at `(p->x, p->y)` with a radius `IRD`, examining potential contacts within its Moore neighborhood across `dx, dy` displacements.

Within the loop, `newly infected` individuals will be identified by their `incubation_days` flag being set to -1 and inserted in the `newly_changed` array.

2) Agent Movement: The Agent Movement phase handles the displacement of each non-deceased agent. A small random displacement in $\{-1, 0, 1\} \times \{-1, 0, 1\}$ is generated.

A check is then performed to ensure that the `new_position (new_x, new_y)` lies within the grid boundaries and that the `occupancy` of the target cell is less than `MAXP_CELL`:

The `movePerson(p, new_x, new_y)` function itself performs two atomic operations: `removePerson(p)`, which removes agent `p` from its current cell and updates the internal cell array to maintain contiguity of `persons`, and `addPerson(p, new_x, new_y)`, which inserts agent `p` into the new cell while updating its `occupancy` counter. If the destination cell is already full, the agent remains in its current position, and no movement occurs.

3) Infectious State Update: The Infectious State Update phase addresses the fate of infected agents whose incubation days reaches 1.

It is important to note that the decrement of `incubation_days` occurs at the beginning of the Disease Spread phase; consequently, checking for a value of 1 precisely triggers the logic on the last day of incubation. Initializing a new infected agent's `incubation_days` to `INCUBATION_DAYS + 1` guarantees that the value remains positive at the very end of the incubation period. This prevents ambiguity with a 0 value, which denotes a non-infected state, thus distinguishing between an agent whose incubation has just concluded and one that was never infected. Once the incubation period concludes, a random probability check against the mortality rate μ determines the agent's outcome. If the random probability is less than μ, the individual recovers; otherwise, they are considered deceased and removed from the map. In the case of recovery, the individual can either gain permanent immunity (setting susceptibility and incubation_days to 0) or enter the "newly recovered" state, marked by `incubation_days` being set to -2. Their pointer is added to the `newly_changed` array for final processing at the end of the daily loop.

Finally, the function concludes with the Commit of New Infections: at the end of the loop over all agents, those individuals marked as *newly infected* or *newly recovered* are updated to *infected* and *susceptible* respectively.

This final commit ensures that `newly infected` individuals become `infectious` and `recovered` individuals become `susceptible` only from the commencement of the subsequent simulation day.

C. Function `main`

The `main` function defines the entry point of the program, orchestrating the entire simulation flow. It begins by invoking `init_population(population)` to prepare the simulation grid and establish the initial states of all agents. Subsequently, it executes the primary simulation loop for `ND` days:

```
for (int day = 0; day < ND; day++) {
simulate_one_day(population);
}
```

Upon the completion of all simulation days, the `main` function ensures proper memory management by freeing all allocated resources, specifically `population`, `occupancy_map`, and `all_persons_pointers`.

V. OpenMP Implementation

The fundamental structure of the OpenMP version largely mirrors that of the serial implementation, being founded on the same three core functions: `main`, `init_population`, and `simulate_one_day`. The underlying simulation logic also remains consistent. However, significant differences arise from the integration of OpenMP's tools and facilities, designed to manage different threads and leverage parallel regions. Therefore, this section will focus on illustrating the modifications made to parallelize the algorithms, rather than reiterating the core simulation logic.

A. Population Initialization (`init_population`)

The very beginning of the function marks the first instance of explicit parallelization.

The `#pragma omp parallel for collapse(2)` directive is used to parallelize the nested loops that initialize the `occupancy_map`. In particular, the directive instructs OpenMP to create a team of threads and distribute the iterations of the for loop among them, collapsing the two nested for loops (for x and y) into a single larger loop structure. As a consequence, each thread will be responsible for initializing a contiguous chunk of cells in the flattened 2D grid.

Following the initialization of the occupancy_map, the function proceeds to distribute the population across the grid. The method for population distribution within init_population has undergone a significant change. Originally, the function would immediately enter a parallel region to spread the population. Inside this parallel block, the primary for loop iterated directly over all NP persons. The intention was for each thread to select the random Tuple from a shared list of available tuples to place a person.

This design, however, presented a significant challenge: race conditions. Multiple threads attempting to concurrently call getRandomTupleIndex (and, eventually, removeTupleAt) from a single, shared TList would lead to unpredictable behavior, yielding an incorrect population placement. While such operations could theoretically be protected by *#pragma omp critical* directives, this would introduce severe serialization, negating the benefits of parallelization and reducing performance.

To circumvent these race conditions and preserve parallelism, an alternative was explored: instead of a shared TList, each thread was given its own private TList (named local_coords in the code). The grid cells (and their corresponding tuples) were distributed as uniformly as possible among these private lists. Each thread would then only select coordinates from its local_coords list, avoiding contention.

However, this improved parallelization introduced a new, subtle problem, represented by the *MAXP_CELL* constant (maximum number of persons per cell). The issue arose because the distribution of cells (and thus local_coords tuples) to threads was not perfectly aligned with the openMP distribution of persons to threads. In other words, it could have happened that a thread worked with more people than it could accommodate in the available places (MAXP_CELL per cell).

To definitively solve this problem without sacrificing parallel efficiency, the final program version includes a serial *pre-distribution phase*.

This sequential region, executed before any thread-specific parallel work for population initialization, calculates the fundamental parameters each thread must know.

cells_per_thread is the array that stores the exact number of grid cells each thread is assigned to manage. The calculation aims for a uniform distribution of the total cells among the threads. If the total number of cells is not perfectly divisible by the number of threads, any remainder cells are then distributed one by one among the initial threads:

```
for (int i = 0; i < NTHREADS; ++i)
    cells_per_thread[i] = (TOT_CELL) /
        NTHREADS + (i < (TOT_CELL) %
        NTHREADS ? 1 : 0);
```

This allocation method effectively simulates the iteration distribution behavior inherent to an OpenMP for loop with default *static* scheduling.

At the same time, people_per_thread contains the exact number of persons each thread is responsible for, assigned proportionally to the number of cells each thread manages. Immune and infected are spread to different threads in the same way.

While this method ensures a proportional distribution of susceptible, immune, and infected, it inevitably leaves remaining unassigned individuals that then deterministically distributed to the threads in sequential order, one by one.

The final part of this preliminary sequential computation calculates the specific range of iterations each thread must perform within the subsequent parallel region by defining arrays of ”offsets.”

```
cell_offset[0] = people_offset[0] = 0;
for (int i = 1; i < NTHREADS; i++)
{
    cell_offset[i] = cell_offset[i - 1] +
        cells_per_thread[i - 1];
    people_offset[i] = people_offset[i - 1]
        + people_per_thread[i - 1];
}
```

In particular, cell_offset and people_offset arrays are fundamental for orchestrating the parallel region. Once threads are deployed, each thread (tid) utilizes these pre-calculated offsets to determine its unique segment of the global data. In other words, each thread independently retrieves its specific starting and ending points, effectively defining its private work boundaries.

```
#pragma omp parallel{
    int tid = omp_get_thread_num();
    int cell_start = cell_offset[tid];
    int cell_end = cell_start +
        cells_per_thread[tid];

    int people_start = people_offset[tid];
    int people_end = people_start +
        people_per_thread[tid];
    // ...
}
```

The cell_start and cell_end values enable each thread to accurately generate its own private set of grid coordinates (tuples). This is achieved by iterating from cell_start up to cell_end, converting the linear cell index i into (x, y) coordinates, and adding them to local_coords.

```
for (int i = cell_start; i < cell_end; i++)
{
    int x = i / H;
    int y = i % H;
    addTuple(local_coords, x, y);
}
```

Here local_coords represents a private array containing the specific cell tuples assigned to a single thread, thereby preventing contention on shared resources.

Concurrently, the people_start and people_end offsets allow each thread to precisely identify and work on its designated range of individuals within the global population array.

```
for (int i = people_start; i < people_end; i
    ++){
    Person *p = &population[i];
    // ...
}
```

The rest of the parallel region follows the same population initialization logic of the serial implementation.

B. One Day Simulation (simulate_one_day) and Locks functions

The simulate_one_day function, unlike init_population, does not incorporate any preliminary serial workload distribution. Instead, it immediately starts with a parallel region. Within this region, the newly_changed array, which was part of the serial implementation, is now privatized for each individual thread.

The core logic for the day's simulation remains consistent with the serial version of the program. The primary adaptation lies in the parallelization of the main for loop that iterates over all NP individuals:

```
#pragma omp for schedule(guided)
for (int i = 0; i < NP; i++)
{
    Person *p = &population[i];
    //...
```

This loop is explicitly scheduled using schedule(guided). This scheduling approach offers the advantage of dynamic workload allocation among threads, where the size of each allocated chunk is exponentially reduced. This proves to be particularly beneficial in this context, as the duration of each iteration is not uniform, but it varies significantly depending on the current state of the individual being processed, making the work distribution irregular.

The main challenge in the parallel version of the program is to replicate the logical implementation of its serial counterpart while resolving all potential race conditions that might arise between threads. It is clear that distinct threads should not concurrently access the same locations within the shared occupancy_map. Such scenarios can manifest, for instance, when one thread is in the process of moving a susceptible individual, while another thread is simultaneously performing an infection check on that same susceptible person because it falls within its IRD. Another problematic situation arises when two different threads attempt to move two distinct individuals into the identical cell slot. These examples of critical race condition are solved by leveraging the omp_lock_t type struct provided by the OpenMP library.

The initialization of these locks is handled by a dedicated function, init_locks, defined outside of the simulate_one_day function, of which we report only a fraction:

```
void init_locks(){
    cell_locks = malloc((long)TOT_CELL *
        sizeof(omp_lock_t));
    //...
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < W; i++)
        for (int j = 0; j < H; j++)
            omp_init_lock(&LOCK(i, j));
}
```

where the macro $\&LOCK(i,j)$ is defined as

```
#define LOCK(x, y) cell_locks[x * H + y]
```

at the beginning of the current parallel file.

These locks are therefore defined as ”tied” to a generic coordinate. In this way, they secure access to the occupancy_map itself without creating a generalized critical section nor globally restricting access to the entire map. They rather allow threads to simultaneously explore and process distinct, separated cells, preserving the benefits of parallelization.

Conversely, the destroy_locks function performs the inverse operation, releasing all allocated lock resources.

In practice, each time an infected person checks a nearby location, the corresponding cell's lock is acquired:

```
omp_set_lock(&LOCK(nx, ny));
```

The lock is then released as soon as the inspection process is complete:

```
omp_unset_lock(&LOCK(nx, ny));
```

Similarly, attention must be paid when handling the movement of different individuals, as this involves manipulating two cells (current and target) concurrently, which is addressed by a specific locking strategy. Starting from the (x1,y1) cell, once a new valid (x2,y2) tuple is generated, the locks are acquired and left in a consistent order to prevent any deadlocks.

```
bool lock_current_first = (x1 < x2) || (x1
    == x2 && y1 < y2);
if (lock_current_first){
    omp_set_lock(&LOCK(x1, y1));
    omp_set_lock(&LOCK(x2, y2));
}
else{
    omp_set_lock(&LOCK(x2, y2));
    omp_set_lock(&LOCK(x1, y1));
}
```

If threads were to acquire locks in an arbitrary order (e.g., Thread A locks cell X then Y, while Thread B locks cell Y then X), a deadlock could occur where each thread holds one lock and waits indefinitely for the other to release the lock it needs. To prevent this, the strict locking order is imposed.

Finally, the concluding section of the method is dedicated to update the statuses of *newly infected* and *newly recovered* individuals, having in mind that each thread processes its own local_newly_changed array.

C. Function main

In the main function, the simulation process is initiated by calling init_population, followed by the execution of the simulation loop for ND days through iterative calls to simulate_one_day. The sole distinction from prior implementations lies in the inclusion of init_locks at the function's outset and destroy_locks at its conclusion.

VI. CUDA Implementation

In this chapter we describe the implementation choices and logical design of the CUDA version of the program. We focus on the aspects that make this version best suited for GPU parallelism, highlighting differences with respect to the serial implementation.

The CUDA version preserves the same high-level simulation logic as the serial code (population initialization, daily cycle with disease spread, movement, and state updates), but reorganizes data structures and computations to exploit the GPU paradigm. In the following implementation, each GPU thread will correspond to a unique person in the simulation.

A. Data structures and CPU-GPU management

Consistent with the other versions of the program, all compile-time parameters (`W`, `H`, `MAXP_CELL`, `NP`, `IRD`, `INCUBATION_DAYS`, `BETA`, `ITH`, `MU`, `IMM`, `INFP`) remain defined in `config.h`. However, significant differences arise in terms of data structures due to the distinct memory access patterns required by the GPU. While the serial and OpenMP code utilize a global pointer-based occupancy map and various arrays of structures, the GPU necessitates simpler structures and contiguous memory access patterns to maximize throughput. Specifically, the following arrays are allocated on the GPU:

- `int *d_x, *d_y`: two arrays of length `NP` storing the current (x, y) coordinates of each agent.
- `int *d_incub`: per-agent incubation-day counters, length `NP`.
- `float *d_susc`: per-agent susceptibility values, length `NP`.
- `int *d_newInf`: per-agent flags (0 or 1) indicating "newly infected" status, used to defer state changes until after the infection kernel completes, length `NP`.
- `int *d_slotIndex`: an array of length `NP`, where for each agent, we place an index in the range $[0..\texttt{MAXP\_CELL}-1]$ indicating the slot inside its current cell, or -1 if the agent is dead or not yet placed.
- `int *d_cellCount`: an array of length $W \times H$ storing the current number of agents in each cell.
- `int *d_cellSlots`: an array of length $W \times H \times$ `MAXP_CELL`. For each cell c, its semantic slots occupy indices ranges $[\,c \times \texttt{MAXP\_CELL}, c \times \texttt{MAXP\_CELL} + (\texttt{MAXP\_CELL} - 1)\,]$, each storing an agent ID or remaining undefined if the slot is empty.
- `curandStatePhilox4_32_10_t *d_states`: an array of random number seeds, one for each thread. It is used to generate random values for agent movement, incubation, recovery, and infection.

All of these device arrays are allocated once at startup, eliminating any dynamic allocation inside kernels. Since all primary data arrays reside directly on the GPU, CPU–GPU memory transfers are inherently minimized, being limited to the initial coordinate generation.

The initial coordinates are generated on the host side using `gen_random_coords`, a function very similar to the one employed in the serial code (which itself utilizes the `TList` structure). The generated coordinates are then copied into the `d_x` and `d_y` device array, before any kernels execution.

Thanks to this strategic minimization of data transfers, we expect that the performance differences between the ”evolution-only” runtime (i.e. excluding data transfers) and the full program runtime will be minimal.

B. Kernel Implementations

Below we describe each major CUDA kernel, its purpose, and how it maps the serial logic onto the GPU parallelism paradigm.

1) `init_population_kernel`: This kernel initializes the population by assigning each agent a random, uniform position (pre-computed by a CPU-based function) and setting that agent's state according to the configuration parameters.

This is a straightforward parallelized kernel that directly mirrors the functionality of its serial counterpart. To assign susceptibility with a Gaussian distribution, we use the built-in function `curand_normal` exploiting the seeds stored inside the already mentioned `curandStatePhilox4_32_10_t`.

2) `buildCellSlots`: This kernel is responsible for populating the data structures that replace the large global occupancy map used in prior versions. In the serial version, each agent was placed into a cell via `addPerson(p, x, y)`, which involved updating a pointer array. On the GPU, conversely, each thread independently computes its own cell index *c*, then performs a single atomic increment operation using `atomicAdd(&d_cellCount[c],1)`. This operation concurrently returns the previous count (`pos`), guaranteeing a unique slot for the agent within that cell in case of race conditions. The thread then proceeds to write its own `tid` to `d_cellSlots[c*MAXP_CELL + pos]` finalising the positioning.

3) `infect_kernel`: This kernel is responsible for the disease spread phase within the simulation. To carry out the infection phase, each thread examines a $(2 \times \texttt{IRD} + 1)^2$ square around its own `(x0,y0)` coordinates. For each neighboring cell c within this radius, the kernel reads `d_cellCount[c]` to determine the number of occupants. It then iterates over the `count` entries in `d_cellSlots[c * MAXP_CELL + s]`, retrieving the thread IDs of the agents inside the checked cell. if a susceptible neighbor inside this cell meets the infection criteria, i.e. `BETA` × `d_susc[i]` > `ITH`, the kernel attempts to mark `d_newInf[i]` using an atomic compare-and-swap operation: `atomicCAS(&d_newInf[i],0,1)`. This `atomicCAS` instruction guarantees that only the first thread that attempts to set the `d_newInf[i]` flag succeeds, preventing any subsequent concurrent attempts to overwrite or conflict with the initial infection mark.

4) `status_kernel`: Within the kernel, each thread primarily performs two operations. If an agent's `d_incub[tid]` value is 1, it means that the agent's final incubation day is over. In this case, a uniform random probability p is drawn via `curand_uniform`: if $p < \mu$, the agent recovers; otherwise, it dies. In the case the agent dies, an `atomicSub(&d_cellCount[c],1)` operation is performed. In case the deceased agent's slot was not the last occupant in its cell, the last occupant is swapped into the newly vacated slot, and its `d_slotIndex` is updated accordingly to maintain a compact list of occupied slots.

5) `move_kernel`: This last kernel is responsible for individuals movement across the cells. This function allows each non-dead thread to initiate its movement by generating two random numbers, `r1` and `r2`, using `curand(&states[tid])`.

To attempt entry into a new cell, `newC`, the thread executes `pos = atomicAdd(&d_cellCount[newC],1)`. If the returned `pos` is less than `MAXP_CELL`, the move is successful. In this case, the thread's `tid` is written into `d_cellSlots[newC * MAXP_CELL + pos]`, its `d_slotIndex[tid]` is updated to `pos`, and its coordinates `(d_x[tid],d_y[tid])` are set to `(newX,newY)`.

Following a successful move, the count of the agent's old cell, `oldC`, is atomically decremented via `oldCount = atomicSub(&d_cellCount[oldC],1)`. If the departing agent's slot was not the last occupant within `oldC`, as in the previous kernel we act with a swap to maintain contiguity.

Conversely, if `pos` is greater than or equal to `MAXP_CELL`, it indicates that the intended new cell is already full. In this scenario, a rollback mechanism is triggered via `atomicSub(&d_cellCount[newC],1)`. This effectively undoes the initial `atomicAdd`, and the agent's coordinates and `d_slotIndex` remain unchanged. This rollback approach was chosen to circumvent the control needed to verify the cell's state between an availability check and the actual increment of its count.

C. Host `main()` Function and Simulation Loop

The host `main()` function orchestrates the CUDA simulation as follows:

1) Allocate host arrays `h_x, h_y` and call `gen_random_coords(h_x, h_y)`, which uses the `TList` structure to generate `NP` random grid positions that respect the occupancy constraint.
2) Copy `h_x, h_y` to `d_x, d_y` via `cudaMemcpy`.
3) Launch `init_curand_kernel(d_states, seed)` to initialize each thread's RNG state.
4) Launch `init_population_kernel(...)` to initialize `d_incub`, `d_susc`, `d_newInf`, and `d_slotIndex`.
5) Launch `buildCellSlots(...)` to populate `d_cellCount`, `d_cellSlots`, and `d_slotIndex`.
6) Main Simulation Loop (for each day):
   a) Launch `infect_kernel(...);`
   b) Launch `status_kernel(...);`
   c) Launch `move_kernel(...);`

   After each of these kernel launches, a `cudaDeviceSynchronize()` call is placed. This ensures that all threads from the current kernel complete their execution before the next kernel is launched. This synchronization is necessary to preserve the logical order of events (infection, status update, movement), avoiding inconsistencies in the simulation.
7) After the loop ends, free all device memory (`cudaFree`) and any remaining host memory.

VII. Report Format

In this chapter we discuss the report format that is produced by the simulation. The result is not simply a single summary file listing who died and who recovered, instead, we designed a report structure that supports a wide range of testing and analysis. This structure allows us to visualize, test, and analyze the simulation outputs from each implementation (serial, OpenMP, CUDA) in a technology-agnostic manner. To achieve this, we have defined an `enum` type and a `struct` as follows:

```
typedef enum
{
    IMMUNE      = 0,
    INFECTED    = 1,
    SUSCEPTIBLE= 2,
    DEAD        = 3
} State;
```

and

```
typedef struct
{
    int   x, y;
    State state;
} PersonReport;
```

Both helps record the agent's state and position within the grid. These two definitions are used by a function called `save_population`, which writes one binary file per day.

Each binary file begins with the first 4 bytes storing an `int` representing `NP`, the total number of agents. This is immediately followed by `NP` × (sizeof(PersonReport)) bytes, which constitute a sequence of `PersonReport` structures. Each `PersonReport` contains three fields: `int x` for the agent's x coordinate, `int y` for the agent's y coordinate, and a `State state` field, which can be one of {`IMMUNE`, `INFECTED`, `SUSCEPTIBLE`, `DEAD`}.

Since these files are in binary format, they cannot be directly opened or interpreted in a standard text editor. However, because all implementations, serial, OpenMP, and CUDA, adhere to the exact same structure, any code designed to read a `day_XXX.dat` file will consistently be able to recover the data.

Saving the full population state to disk every day can be time-consuming, especially when `NP` is large. Therefore, we wrap the `save_population` call in a conditional section, controlled by a `debug` flag:

```
for (int day = 0; day < ND; ++day){
    if (debug)    {
        save_population(d_x, d_y, d_incub,
            d_susc, day);
    }
    // ... rest of the daily simulation ...
}
```

In the CUDA code (and similarly in the serial/OpenMP code), this means that `save_population` is invoked only when `debug == true`. Otherwise, the simulation runs without writing any daily reports.

Writing a full report each day cause a significant I/O overhead, whereas a minimal end-of-run summary would have a negligible impact on runtime. For this reason, daily reporting is optional and only enabled when detailed testing or analysis is required so to not influence the performance test.

VIII. Python Visualizer and Unit Testing

Using the reports mentioned above, we developed a visualizer for the simulation to provide a qualitative overview of the behavior. The visualizer is implemented as a concise Python script; we chose this language for its brevity and powerful libraries such as `matplotlib`.

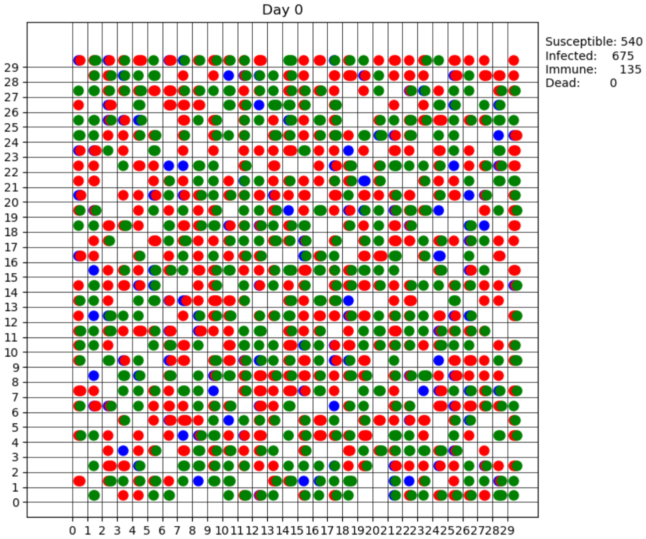


Fig. 3: Visualizer at Day 0: initial distribution of susceptible (green), infected (red), immune (blue), and (none yet) dead (black).

Also another script was implemented to verify correctness and robustness of the different program versions. We used Python's `unittest` framework to read each binary report file (`day_XXX.dat`) and check the project. This ensures that all implementations (serial, OpenMP, CUDA) produce consistent, valid outputs.

A. Testing Strategy

Our tests perform the following checks:

- **File Existence & Header.** Confirm that each expected file exists and its header integer equals `NP`.
- **Correct Number of Entries.** Ensure each file contains exactly `NP` records.
- **State Count Consistency.** the totality of all agent in state {`IMMUNE`, `INFECTED`, `SUSCEPTIBLE`, `DEAD`}; must be equal to `NP`.
- **Occupancy Limit.** Group living agents by their (x, y) coordinates and verify no cell has more than `MAXP_CELL` occupants.
- **Valid Positions or Dead Flag.** Each (x, y) must lie in $[0..W-1] \times [0..H-1]$ or be $(-1, -1)$ if dead.
- **One-Cell Movement per Day.** For each agent, compare positions on day $d-1$ versus day d. If both are valid, enforce $|x_d - x_{d-1}| \leq 1$ and $|y_d - y_{d-1}| \leq 1$.
- **Monotonic Deaths.** The cumulative number of dead agents must never decrease from one day to the next.
- **Initial Immune/Infected Counts.** On day 0, immune count must equal $\lfloor NP \times IMM \rfloor$, and infected count must equal $\lfloor NP \times INFP \rfloor$.
- **Incubation Duration & Transition.** Each agent's sequence of `INFECTED` states cannot exceed `INCUBATION_DAYS`. Immediately after that run, the next state must be `IMMUNE`, `SUSCEPTIBLE`, or `DEAD`.

By grouping these checks in a concise test suite, we automatically detect missing files, incorrect agent counts, illegal movements, capacity violations, and invalid state transitions without manually inspecting dozens of scenarios.

IX. Performance analysis

In this section, we report the results concerning the performance comparison between the three implementations. To avoid overloading the section with too many plots and to keep the discussion concise, we fixed all logical parameters and varied only the total number of persons (`NP`) and the side length of the 2D simulation grid. The default parameters were set at $\texttt{MAX\_PCELL} = 3$, $\texttt{IMM} = 0.1$, $\texttt{INFP} = 0.5$, $S_{\text{AVG}} = 0.5$, $\Delta S = 0.1$, $\texttt{ND} = 20$, $\texttt{INCUBATION\_DAYS} = 4$, $\beta = 0.8$, $\texttt{ITH} = 0.2$, $\texttt{IRD} = 1$, $\mu = 0.6$.

We defined the population density as:

$$n = \left\lfloor \frac{NP}{W \times H \times \text{MAXP\_CELL}} \right\rfloor \tag{2}$$

and chose to keep it fixed at three representative values: 10%, 50%, and 90%, following the "10-50-90 rule". For each of these densities, we varied the grid dimensions from a minimum of 10×10 m^2 to a maximum of 10000×10000 m^2 and obtained NP by reverting (2).

To ensure statistically meaningful results, each simulation was executed multiple times (`NRUNS` = 10), and the reported execution times were averaged over these runs afterwords. Submissions were automated using the scripts `generate_submissions_serial.sh`, `generate_submissions_OMP.sh`, and `generate_submissions_CUDA.sh`, which produced jobs for a range of (W, n) values. Simulations were run on Polito's HPC infrastructure using the Hactar cluster. On Hactar, each node provides 24 physical CPU cores. To analyze performance scaling, we varied the number of threads across powers of two up to the hardware limit. Specifically, in the OpenMP submission scripts, we modified:

- `export OMP_NUM_THREADS=n` for n =2, 4, 8, 16, 24
- `#SBATCH cpus-per-task=n` accordingly

For the CUDA implementation, we kept the same script and just modified the line `int threads = n;`, for $n =$

$128, 256$ and 1024, with the number of blocks that will vary both as a function of `threads` and the number of persons NP according to the parametric relation `int blocks = (NP + threads - 1) / threads;`.

For every implementation and every density, output files of the run $i$, called `run$i$.out` have been ordered in folders of the form `results_W_NP`. Each file contains the output of a print of the form `RunTime: % ms`, with the exception of the Cuda one, where we have distinguished between the execution time including the host-device memory exchange (`RunTime`) and that only spent during GPU kernel execution (`RunTime_Evo`). As explained in section (VI), since our Cuda version creates and elaborates all the data (with few exceptions) in the device, we don't expect any difference between cuda and cuda evo. This expectation is confirmed by the output files, which show maximum discrepancy of around 1 % across all tested problem sizes. We will henceforth refer to the overall CUDA runtime without further distinguishing between these two measurements.

With the help of a python script, we read the times in ms from the outputs and we plotted them against the grid size for the three values of density n. The results are shown in figures 4, 5 and 6.

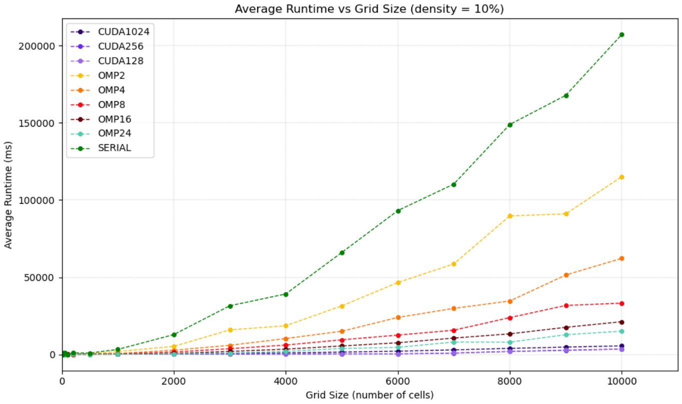


Fig. 4: Plot of execution time versus grid dimension for density $n = 0.1$

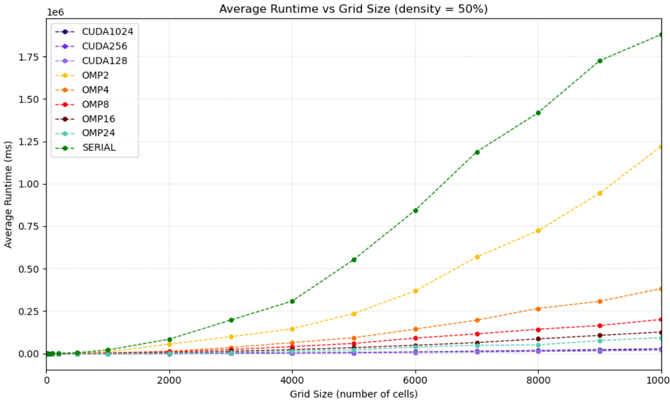


Fig. 5: Plot of execution time versus grid dimension for density $n = 0.5$

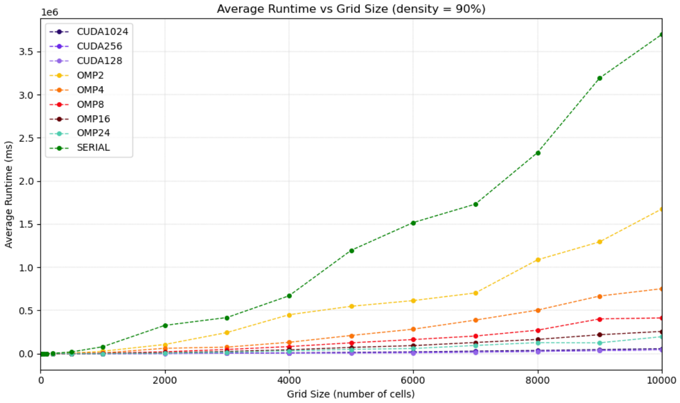


Fig. 6: Plot of execution time versus grid dimension for density $n = 0.9$

Results show the increase of execution time with larger grid sizes for the different population densities; as expected, with growing number of persons all the implementations take more time to run.

If we focus on the largest grid size (10000×10000), we can quantify how sensitive each implementation is to increasing density, by computing the average rate of increase.

TABLE I: Average runtimes (ms) for different implementations at densities 0.1 and 0.9 at $W = 10000$ and corresponding time ratios.

Implementation	$t_{n=0.1}$ [1e4 ms]	$t_{n=0.9}$ [1e4 ms]	$\frac{t_{TOT,n=0.9}}{t_{TOT,n=0.1}}$
SERIAL	20.71	369.80	16.9
OMP2	11.51	167.56	13.6
OMP4	6.22	75.19	11.1
OMP8	3.32	41.33	11.4
OMP16	2.12	25.70	11.1
OMP24	1.51	19.60	12.0
CUDA1024	0.56	5.70	9.2
CUDA256	0.35	4.50	11.9
CUDA128	0.36	4.40	11.2

As depicted on table (I), all parallel versions are more robust with respect to the serial one versus the increase in population size at fixed grid dimension, with an average increase of a factor of order 10 in all implementations and the best one being CUDA with 1024 threads.

Finally, to evaluate the effective speedup achieved by the parallel implementations, we compared the ratio between the runtime of the serial version and the corresponding parallel one. To avoid cluttering the report with too many figures, we report only the plot corresponding to the case $n = 0.9$, which is representative of the general behavior observed across all densities. Moreover, we restrict our analysis to grid sizes greater than 4000, since we are interested in the asymptotic behavior. For smaller grids, thread generation and work distribution overheads dominate the computation time, masking any potential speedup benefits. For comparison purposes, we consider the time ratio corresponding to $W = 10000$ and we evaluate it against the standard *Amdahl law*, which states that

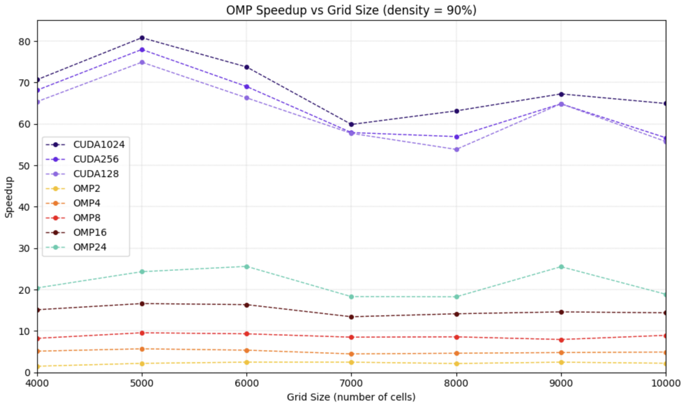


Fig. 7: Plot of relative speedup of the parallel implementations against the serial one for all OMP and CUDA versions expressed as ratio of average runtimes.

for a parallel program with N independent threads:

$$t_{\text{TOT}}(N) = t_{\text{serial}} + \frac{t_{\text{parallel}}}{N} \qquad (3)$$

where t_{serial} refers to the serial portion of the code, which cannot be reduced using more than 1 thread and t_{parallel} refers to the one that can be reduced using more threads.

From this law, we expect the total speedup for all parallel versions to be upper-bounded by the total time of the serial divided by the number of threads. The results of the analysis are depicted in the next table.

TABLE II: Comparison between actual speedup and expected improvement number of threads for all parallel versions of the code ($W = 10000,\ n = 0.9$).

Implementation	Speedup	Number of Threads
OMP2	2.2	2
OMP4	4.9	4
OMP8	8.9	8
OMP16	14.4	16
OMP24	18.9	24
CUDA1024	64.92	1024 × 263672
CUDA256	56.65	256 × 1054688
CUDA128	55.74	128 × 2109376

The OpenMP implementations generally follow the expected linear trend in speedup with increasing thread count. In particular, thanks to the use of locks which minimize the serial portion of the code, we expect $t_{\text{serial}} \ll t_{\text{parallel}}$ and therefore $t_{\text{TOT}}(N) \approx \frac{t_{TOT,SERIAL}}{N}$. Minor deviations can be attributed to the stochastic nature of the simulations (e.g., the random placement and movement of individuals) as well as to runtime fluctuations due to machine-level factors. CUDA implementations demonstrate a significant speedup, thanks to the huge number of threads allowed by the GPU. We observe no significant differences in the speedup among the three different choices of number of blocks and threads, since (as we expect from our programming choice) the implementation only takes advantage of the total number of threads equal to NP and not make use of the different number of blocks explicitly[2].

In summary, the performance gains across the implementations match the theoretical expectations: OpenMP shows nearly linear scaling with the number of threads, with slight saturation beyond 16 threads, while CUDA achieves the best overall speedups—reaching a nearly constant factor of $\sim$ 60—regardless of the specific thread-per-block configuration.

X. Simulation Analysis - Plots

In this final section, as required by the assignment proposal, we analyze how the evolution of the population changes as a function of the logical parameters of the simulation. Since we were not interested in performance aspects, we decided to use OpenMP with 8 threads. In particular, we fix the input parameters as follows:

$$ND = 30,\ W = H = 1000,\ \texttt{MAX\_PCELL} = 3,\ n = 0.9 \qquad (4)$$

We also maintain the same initial conditions across all simulations, to ensure comparability of the results:

$$\texttt{INFP} = 5\%, \texttt{IMM} = 1\%, \texttt{INCUBATION\_DAYS} = 4, \texttt{IRD} = 1 \qquad (5)$$

The main parameters that characterize the infection dynamics and can be externally varied are μ, β, S_{AVG}, and ITH. However, the last three are interdependent, and the only relevant combination for the simulation behavior is:

$$\frac{S_i\beta}{ITH} \overset{?}{>} 1 \quad \text{or} \quad \frac{S_i\beta}{ITH} \overset{?}{<} 1 \qquad (6)$$

Hence, it suffices to fix two of the three parameters in Eq. (6) and vary the third. We choose to vary S_{AVG}, while keeping fixed the standard deviation $\Delta S = 0.1$ and the other two parameters at:

$$\beta = 0.8 \quad ITH = 0.5 \qquad (7)$$

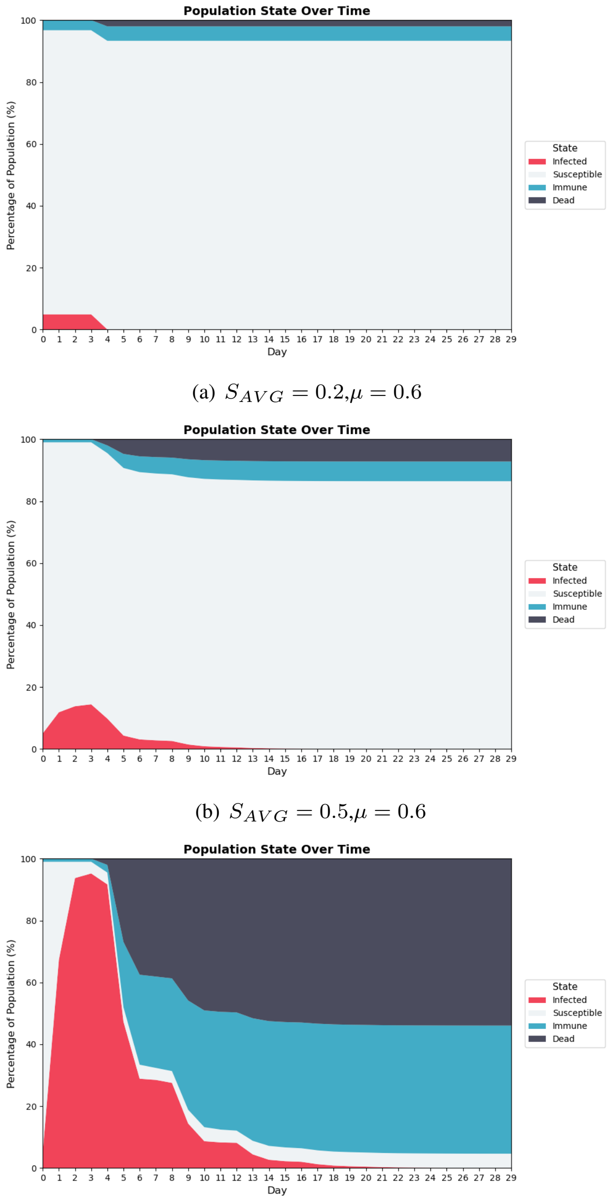
The main question we now address is: *Which parameter is more critical for the evolution of the population — μ or S_{AVG}?* To investigate this, we performed a series of simulations varying $S_{\text{AVG}} \in \{0.2, 0.5, 0.8\}$ while fixing $\mu = 0.6$, and vice versa, varying $\mu \in \{0.3, 0.6, 0.9\}$ with $S_{\text{AVG}} = 0.7$[3].

For each of the 6 possible configurations of parameters, we run the simulations and with the help of the python script `plot_logical_test` we displayed the population evolution with time, i.e. how the percentage of infected, susceptible, immune and dead persons changes as days pass by. The results are depicted in figures (8) and (9).

As clearly shown in Figure 8, since we fixed the ratio $\frac{ITH}{\beta} = 0.625$, average susceptibility values of 0.2 and 0.5 result in a negligible number of deaths and newly immune individuals. This is because, in most cases, the infection condition (1) is not satisfied. In contrast, when $S_{AVG} = 0.8$ (Figure 8(c)), we observe a sharp increase in the number of

[2] Namely, our implementation only exploits the global memory and does not take advantage of the fast data exchange capability enabled by the shared memory of blocks.

[3] The default parameters in the original assignment were such that they led to an uninformative simulation: the number of infectious individuals dropped to zero within a few days, as the condition (1) for new infection was almost never satisfied.

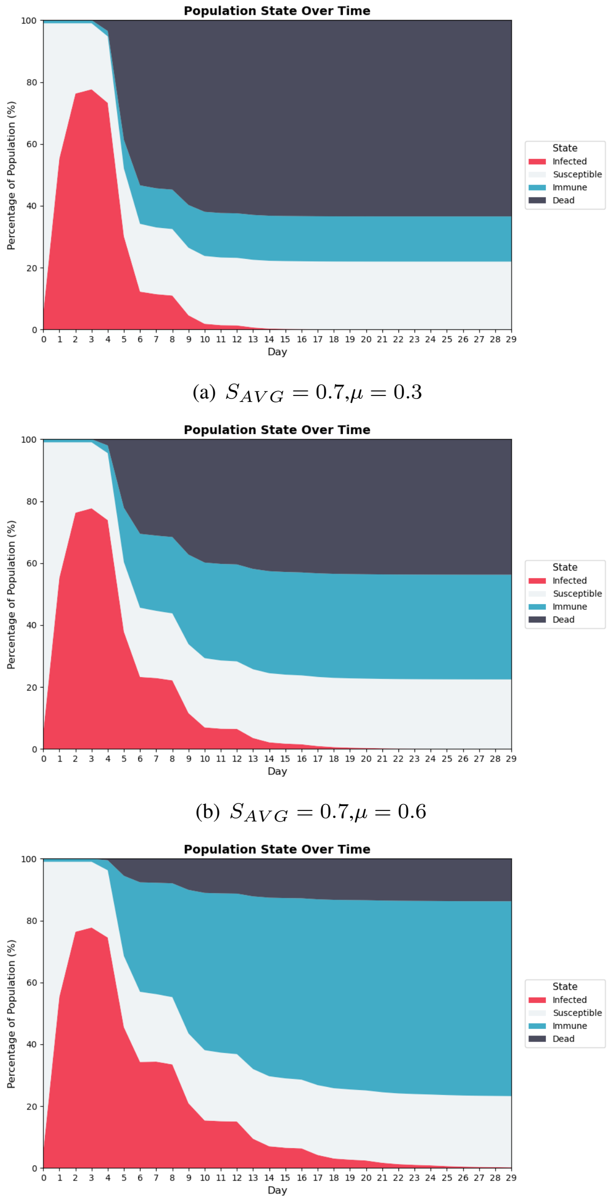


(a) $S_{AVG} = 0.2$,$\mu = 0.6$

(b) $S_{AVG} = 0.5$,$\mu = 0.6$

(c) $S_{AVG} = 0.8$,$\mu = 0.6$

Fig. 8: Simulation results for varying average susceptibility.



(a) $S_{AVG} = 0.7$,$\mu = 0.3$

(b) $S_{AVG} = 0.7$,$\mu = 0.6$

(c) $S_{AVG} = 0.7$,$\mu = 0.9$

Fig. 9: Simulation results for varying recovery probability.

infected individuals during the initial days of the simulation, as nearly all susceptible individuals become infected. This is followed by a decline in infectious persons, leading to a stable outcome where the population is composed mostly[4] of immune individuals and a few remaining susceptibles with small susceptibility.

Vice versa, looking at figures 9, we observe a smoother increase in the number of immune persons with consequent decrease in the number of dead persons from 9(a) to 9(c), as expected. An additional side effect is the slower decrease in the number of infected persons with increasing μ, because 45% of the infected persons will recover and become newly susceptible, and may undergo many cycles of newly infected-newly susceptible before reaching the equilibrium conditions.

To answer the original question, we may say that while an increase in μ smoothly affects the amount of dead and immune persons due to the final recovery check, the behavior of the epistemic dynamics versus S_{AVG} is more threshold-like: fir fixed ratio $\frac{ITH}{\beta}$, values of S_{AVG} below that threshold will lead to an evolution where most persons remain alive, rarely getting infected, whereas larger values of S_{AVG} will lead to a behavior in which almost the whole population is either dead or immune, with a percentage ruled by the recovery

[4] After the incubation period, all surviving individuals either become immune or revert to their original susceptibility level, quickly becoming reinfected. Consequently, the system evolves toward a limiting state dominated by immunes and the small fraction of low-susceptibility individuals who remain uninfected.

probability μ. The sharpness of this transition also depends on the standard deviation of susceptibility (in our simulations fixed at $\Delta S = 0.1$): a smaller standard deviation amplifies this "all-or-nothing" behavior, while a larger one softens it.

In all cases, the limiting behaviors emerge only after few days, as the number of infected persons after an initial spike soon gets back to zero, in which case the interesting part of the simulation ends. Thus, we may conclude that in all proposed cases the epidemic is *self-limiting*, as any infected individuals either die or become immune in a way that prevents further spread.

XI. Conclusions

This project demonstrated how parallel computing techniques can effectively accelerate agent-based epidemic simulations. The serial, OpenMP, and CUDA implementations preserved a common simulation logic while achieving increasingly better performance. OpenMP exploited CPU parallelism with careful synchronization, while CUDA offered the highest speedup through massive GPU parallelism and optimized memory handling.

The modular design, reporting system, and visualizer allowed for comprehensive testing and intuitive analysis of the simulation. Both performance and logical analyses confirmed the validity of the model, aligning with theoretical expectations and reinforcing its value as a predictive tool for studying disease spread in epidemiological contexts.

References

[1] W. O. Kermack and A. G. McKendrick, "A contribution to the mathematical theory of epidemics," *Proceedings of the royal society of london. Series A, Containing papers of a mathematical and physical character*, vol. 115, no. 772, pp. 700–721, 1927.