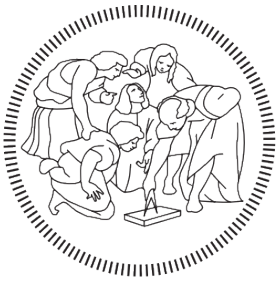


POLITECNICO
MILANO 1863

POLITECNICO DI MILANO

MASTER'S PROGRAM IN
HIGH PERFORMANCE COMPUTING ENGINEERING

High-Performance Parallel Implementation of the Susceptible-Infected-Recovered (SIR) Epidemic Model using MPI and Runge-Kutta Numerical Integration

Group Members:

Milica Sanjevic	Person Code: 10975337
Salvatore Mariano Librici	Person Code: 11078653
Yibo Li	Person Code: 11022291
Nada Elsayed	Person Code: 10998973
Hirdesh Kumar	Person Code: 10997383

Academic Year:
2024/2025

Contents

Abstract: This project presents a parallel simulation framework based on the classical SIR (Susceptible-Infected-Recovered) epidemic model, aimed at studying the dynamics of disease spread in spatially structured populations. The model discretises the simulation domain into a 2D grid where each cell evolves over time according to the SIR equations. To ensure numerical accuracy, the system is integrated using the 4th-order Runge-Kutta method. For scalability, the implementation employs MPI-based domain decomposition and ghost cell exchange, allowing the simulation to run efficiently across multiple processors. Experimental results validate the correctness and performance of the framework, and the architecture is designed to be extensible for other spatial epidemic or information propagation scenarios.

1.1 Introduction

Modeling the spread of diseases across spatial domains is an important task in computational epidemiology. Classical compartmental models, such as the SIR (Susceptible-Infected-Recovered) model, capture population-level transitions between health states, but often assume a well-mixed population. In reality, geographic and social structures cause spatial heterogeneity, making grid-based modeling more appropriate for certain applications.

In this work, we simulate the SIR model over a two-dimensional spatial grid, where each cell represents a subregion of the population. The infection dynamics are governed by ordinary differential equations, which are numerically solved using a 4th-order Runge-Kutta (RK4) integration scheme to ensure accuracy and stability.

To enable high-resolution and large-scale simulations, we parallelize the framework using the Message Passing Interface (MPI). The spatial domain is decomposed across MPI ranks, and ghost cell communication is used to preserve local interaction at region boundaries. The simulation is implemented in C++, and performance evaluations confirm its scalability and balanced workload distribution across ranks.

We adopt MPI to support distributed memory parallelism, which is essential for scaling to large domains and executing the simulation efficiently on multi-node systems.

Although this work focuses on infectious disease modeling, the underlying architecture can be generalized to simulate other types of spatial propagation processes, such as information diffusion or network congestion, making it a versatile foundation for future research.

1.2 Background and Definitions

Grid Structure and Neighborhoods. The simulation domain is discretized into a two-dimensional grid, where each cell represents a subregion of the population. For each cell, we define a neighborhood consisting of the adjacent cells (typically using the 4-neighbor or 8-neighbor scheme) to model spatial interactions in the infection dynamics.

Ghost Cells and Parallelization. In the MPI-based parallel implementation, the grid is partitioned across multiple ranks. To maintain data consistency at subdomain boundaries, we use ghost cells—copies of neighboring cells from adjacent ranks—which are updated through inter-process communication at each time step.

Single Cell Evolution. Each simulation cell independently evolves over time according to the SIR model's ordinary differential equations (ODEs). The evolution is computed using a 4th-order Runge-Kutta (RK4) method for numerical integration. In the parallel version, each RK4 step incorporates values from neighboring cells via ghost cell communication to model spatial infection spread.

1.3 Mathematical Formulation

We adopt the classical SIR (Susceptible–Infected–Recovered) model to simulate the dynamic evolution of states within a spatial grid. Each grid cell maintains the proportion of its population in the three states:

- **S**: Susceptible individuals
- **I**: Infected individuals
- **R**: Recovered individuals

The local dynamics of state transitions are governed by the standard system of differential equations:

$$\begin{aligned}\frac{dS}{dt} &= -\beta S \cdot I \\ \frac{dI}{dt} &= \beta S \cdot I - \gamma I \\ \frac{dR}{dt} &= \gamma \cdot I\end{aligned}$$

To model spatial interaction across neighboring cells, we extend the equations by incorporating a coupling term that accounts for the influence of infected populations in adjacent locations. The modified equation for the susceptible population in cell (i, j) becomes:

$$\frac{dS_{ij}}{dt} = -\beta S_{ij} \cdot \left(I_{ij} + \sum_{(k,l)\in N(i,j)} I_{kl} \right)$$

where $N(i, j)$ denotes the set of neighboring cells of (i, j). Similar formulations are used for $\frac{dI_{ij}}{dt}$ and $\frac{dR_{ij}}{dt}$ to capture spatially distributed SIR dynamics.

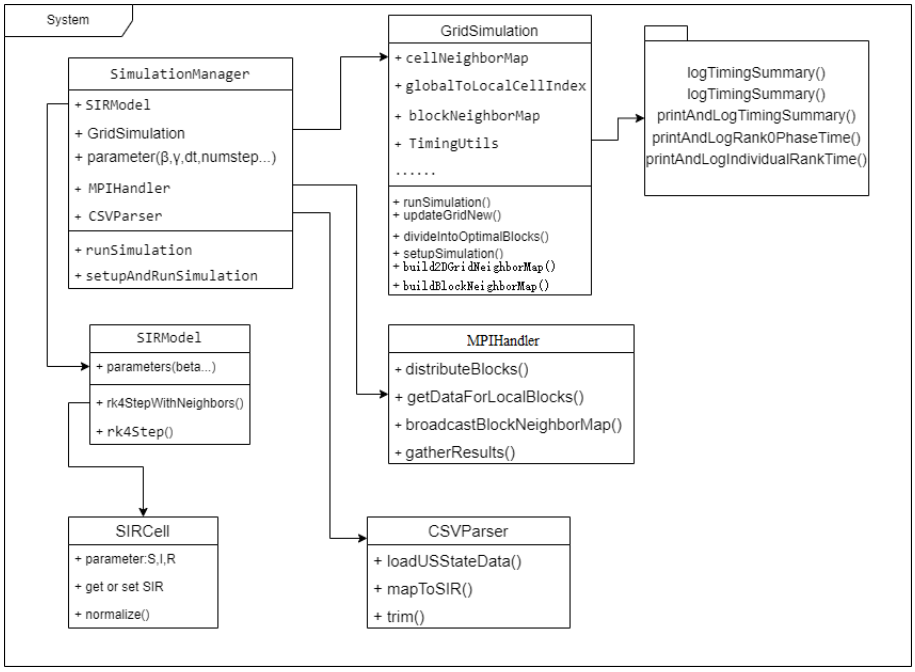
1.4 Numerical Methods

We employ a 4th-order Runge-Kutta (RK4) integrator for each time step to solve the system of ordinary differential equations governing the state transitions.[1] This method offers a good balance between computational efficiency and numerical accuracy, making it well-suited for time-dependent simulations of interacting systems. Compared to simpler schemes such as Euler's method, RK4 significantly reduces error accumulation over long-term integration.

Each cell's state—represented by the fractions of susceptible (S), informed (I), and blocked (R) vehicles—is updated based on both local values and the influence from neighboring cells. To achieve this, we implement a neighbor-aware RK4 variant in the function `rk4StepWithNeighbors()`, which calculates intermediate derivatives using the average infected fraction from adjacent cells. This allows the simulation to realistically capture the spatial diffusion of information or congestion.

After each RK4 update, we apply a normalization step to ensure that the sum $S + I + R = 1$, preserving the physical consistency of the model. This normalization also helps prevent numerical drift due to floating-point approximations across iterations.

1.5 Code Architecture and Structure



1.5.1 CSVParser Class

The `CSVParser` class is responsible for reading and preprocessing real-world COVID-19 data from CSV files. It extracts relevant fields such as population, confirmed, recovered, and active cases, and converts them into normalized SIR values. The parsed data is returned as `SIRCell` instances, ensuring that each cell satisfies $S + I + R = 1$ through internal normalization.

1.5.2 SIRCell Class

This class encapsulates the state variables of each simulation cell: susceptible (S), infected (I), and recovered (R). It provides getter and setter methods with bounds checking to ensure physical validity. After each update, we apply a `normalize()` method that rescales the values so that $S + I + R = 1$.

This normalization is not part of the classical SIR model itself, but a design choice introduced in our implementation to mitigate numerical drift and maintain probabilistic consistency.

1.5.3 SIRModel Class

The `SIRModel` class defines the evolution equations and implements two versions of the Runge-Kutta 4th-order integration: one for isolated cells, and one that incorporates neighbor influence (`rk4StepWithNeighbors`). It holds model parameters such as β, γ, timestep size, and total number of steps.

1.5.4 GridSimulation Class

This class is responsible for managing the entire simulation grid, including the local cells, global-to-local ID mappings, and ghost regions. It handles the step-by-step evolution of the system, sets up the neighbor relations, and maintains grid partitioning and block ownership across ranks.

1.5.5 TimingUtils

The `TimingUtils` provides utility functions for logging and analyzing the runtime performance of the simulation. It records the execution time of each phase (e.g., initialization, communication, computation) across all MPI ranks. The class supports generating both global and rank-specific timing summaries, including minimum, maximum, and average statistics. This data helps identify bottlenecks and evaluate the scalability of the simulation framework.

1.5.6 MPIHandler and SimulationManager Classes

The `MPIHandler` class abstracts all MPI-related communication, including distributing grid blocks, exchanging ghost cells, and gathering final results. The `SimulationManager` class coordinates the entire simulation workflow: loading data, initializing components, running the main loop, and storing output.

1.5.7 Design Principles and Code Organization

The code is structured with modularity and scalability in mind. Core logic is separated from communication routines, and all timing-related functionality is encapsulated in a utility module. The design ensures clear responsibility separation and allows easy testing or extension (e.g., for other ODE models or input formats).

1.6 Parallelization Techniques and Implementation

1.6.1 Parallelization with MPI

To achieve scalable simulation of information and congestion propagation on large-scale smart road systems, this project employs the Message Passing Interface (MPI) to distribute the computational workload across multiple processors. Each processor (MPI rank) is responsible for evolving a subset of the overall grid representing spatial regions (e.g., states or road cells). Inter-process communication is used to exchange boundary (ghost) cells, enabling accurate interaction across adjacent regions.

MPI was selected due to the following advantages:

- **Scalability**: MPI allows distributing computation across multiple cores and nodes, making it suitable for simulating large grids (e.g., 50+ cells).
- **Explicit communication**: As our model requires interaction between neighboring cells, MPI's explicit control over data exchange provides both flexibility and performance.
- **Process isolation**: Each process handles only its assigned cells, reducing memory contention and increasing cache efficiency.

MPI Parallelization Workflow

The main simulation procedure using MPI consists of the following steps:

1. **Grid Partitioning (Rank 0)**: The global grid is first mapped using a cell ID mapping loaded from a CSV dataset. The function `divideIntoOptimalBlocks()` chooses the most balanced partitioning scheme. Cells are grouped into blocks and assigned to ranks.
2. **Initial Communication**: Rank 0 broadcasts:
   - Block-to-rank map
   - Cell neighbor relationships
   - Ghost neighbor relationships

   Each rank initializes its local SIRCell grid using `setGridFromLocalData()`.
3. **Ghost Cell Communication**: During each time step, ghost boundary cells are exchanged using non-blocking MPI operations:
   - `MPI_Isend` and `MPI_Irecv` transfer S, I, R values to/from neighboring ranks.
   - `MPI_Barrier` ensures synchronization.

- Mapping from local index → global ID and global ID → owner rank is used to build send/receive buffers.

4. **Local Update**: Each cell is updated using `rk4StepWithNeighbors()`, incorporating the influence of both local and ghost neighbors. A normalization step ensures $S + I + R = 1$ for every cell.

5. **Result Collection**: At the end of the simulation, each rank outputs its local average S, I, R to a time-series vector. Optionally, Rank 0 gathers global results for visualization.

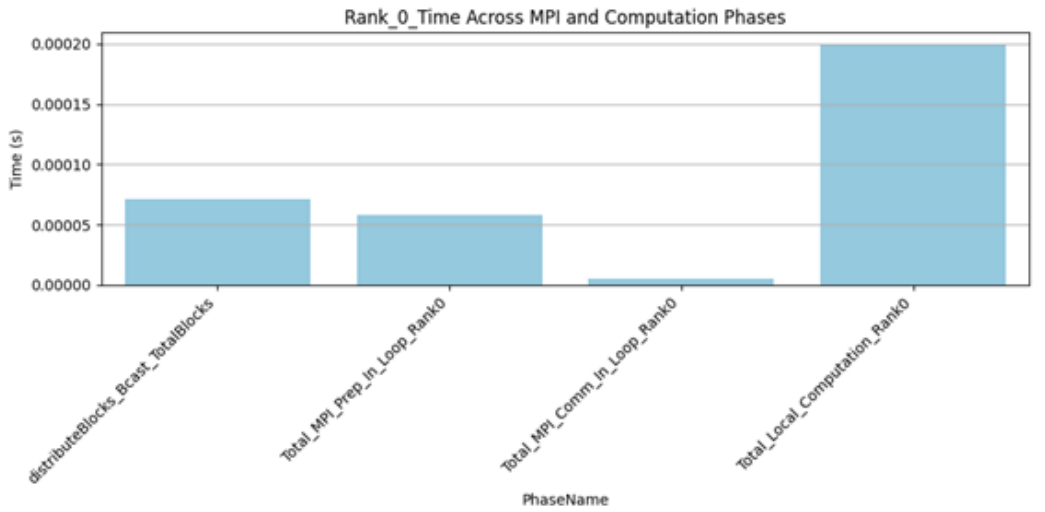
1.6.2 MPI Functionality Highlights

To understand performance behavior and potential bottlenecks, we conducted fine-grained timing analysis across different phases. Each rank logs the time spent in key MPI-related sections using a custom timing utility module (`TimingUtils.cpp`). This provides rank-specific insights into load balancing and communication overhead.

Phase	Description
`distributeBlocks()`	Initial grid partitioning on rank 0
`Total_MPI_Prep_In_Loop_RankX`	Time preparing ghost data buffers per step
`Total_MPI_Comm_In_Loop_RankX`	Actual non-blocking send/receive time
`Total_Local_Computation_RankX`	RK4 updates using local + ghost values
`runSimulation_TotalWallTime`	Overall simulation time (barrier-to-barrier)

Table 1.1: MPI Timing Categories

We observed that computation dominates total execution time, but MPI communication becomes significant as the number of ranks increases.

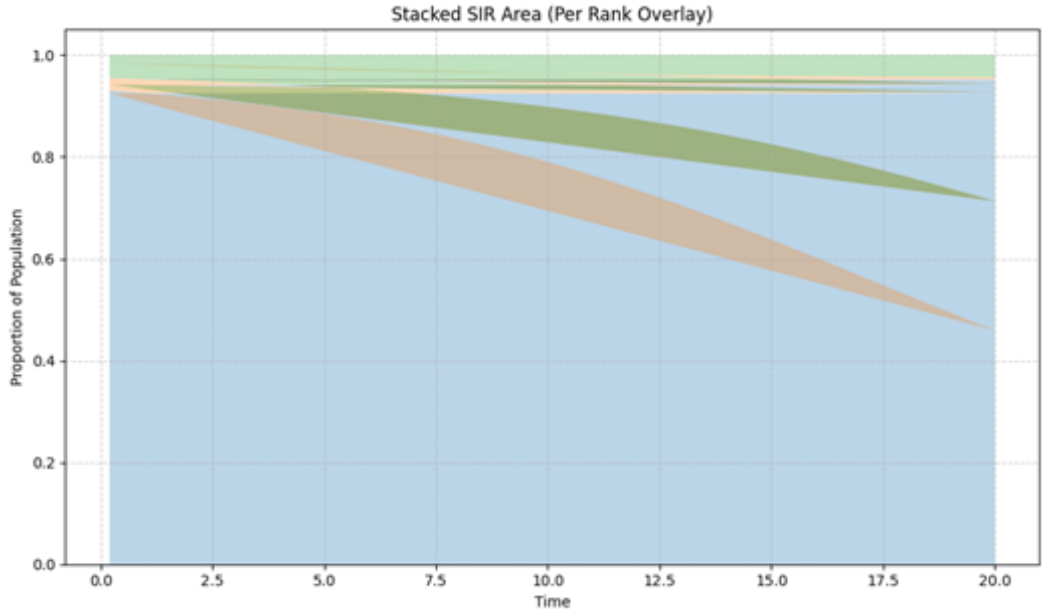


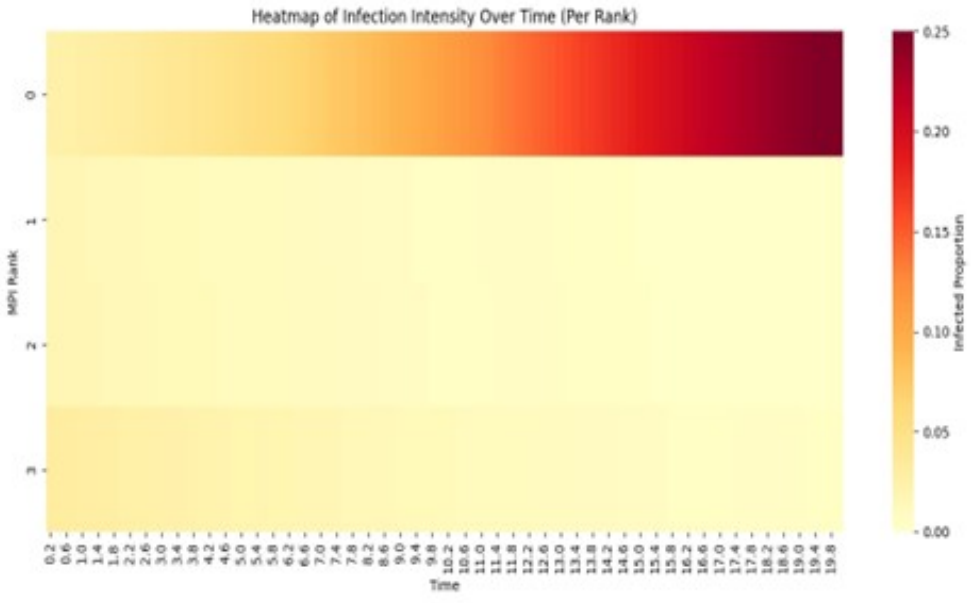
1.6.3 Visualization of Spatial Communication

To explore the impact of rank-based spatial decomposition, we visualize the evolution of infection intensity per rank over time. Each MPI rank controls a specific grid region (e.g., a block of US states), and infection spreading dynamics are influenced by local conditions and ghost exchange.

Key observations:

- Some ranks remain uninfected due to lack of spatial contact (e.g., isolated blocks).
- Peak infection times vary across ranks due to asynchronous propagation.



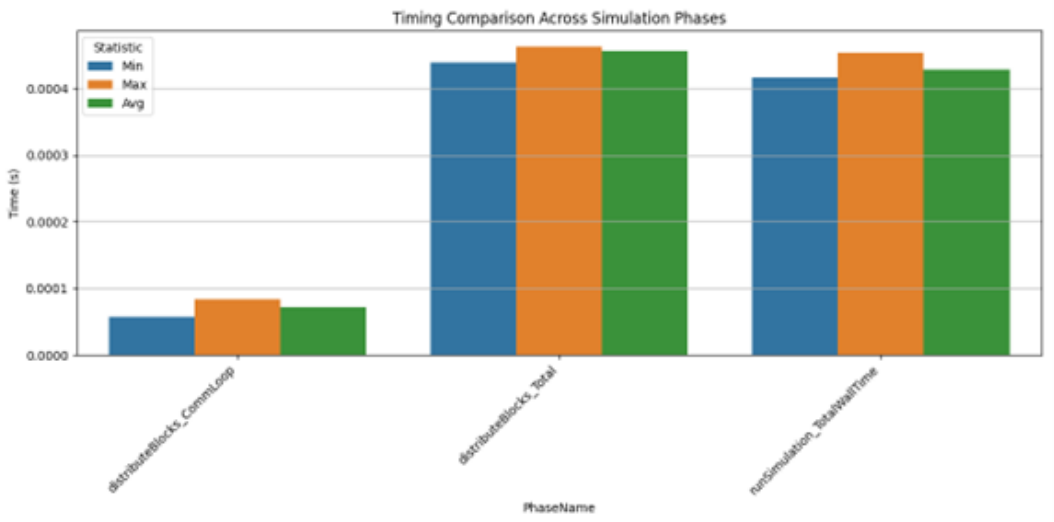


1.6.4 Performance Results and Scalability

We compared execution time across configurations (e.g., 4 ranks, 8 ranks). As expected, total wall time decreases with more ranks, up to a limit. Communication overhead grows superlinearly due to ghost cell exchange.

Observations:

- For 4 ranks, communication is negligible compared to computation.
- Communication overhead grows with process count.
- Load balancing is maintained via `divideIntoOptimalBlocks()`.



1.6.5 Summary of parallel implementation

The MPI-based simulation achieves efficient spatial parallelization of the SIR model by distributing blocks of the grid across multiple processes. Each MPI rank is responsible for computing a subset of the grid, and ghost cells are exchanged between neighboring ranks to ensure correct interactions at boundaries.

This parallel structure reduces computation time significantly for large grids while preserving accuracy in local dynamics through explicit message passing. The simulation is organized in a modular fashion, separating responsibilities for grid partitioning, communication, and local updates, which improves code maintainability and profiling capabilities.

1.7 Experimental Setup

1.7.1 Input Data

The simulation is initialized using real-world data derived from U.S. COVID-19 state-level daily reports, published by the Johns Hopkins CSSE dataset. Each record corresponds to a U.S. state and includes epidemiological and demographic features.

Before integration, the raw data is cleaned and preprocessed using Python scripts (`clean_sort_dataset.py`). The resulting CSV files contain:

- `Province_State`: Name of the U.S. state
- `Population`: Total number of residents in the state
- `Lat, Long`: Geographic coordinates
- `Confirmed, Deaths, Recovered`: Historical COVID-19 case data
- Derived S, I, R values based on these statistics

Each file is reformatted with consistent headers and column ordering. Missing values are filled using backward filling or known approximations. The data is sorted geographically to preserve adjacency relationships in the simulation grid.

Example files: `sorted_01-01-2021.csv`, `sorted_02-05-2021.csv`

1.7.2 Simulation Parameters

We test the model under various combinations of infection rate β and recovery rate γ to reflect different behavioral and traffic conditions. These parameters mirror classical SIR dynamics:

- β: Controls how fast information spreads among vehicles
- γ: Reflects how quickly congestion is resolved

Scenario	β	γ	**Interpretation**
Baseline	0.3	0.1	Moderate info spread and recovery
Fast Spread	0.5	0.1	Aggressive dissemination of signals
Fast Recovery	0.3	0.3	Faster clearing of blocked roads
Low Transmission	0.1	0.1	Weak signal sharing; slow dynamics

Table 1.2: Tested parameter scenarios

Each configuration simulates 20 iterations using a fixed time step of $\Delta t = 0.2$.

1.8 Results and Analysis

1.8.1 Accuracy and Model Behavior

The model enforces the constraint $S + I + R = 1$ at every grid cell and time step, ensuring physical consistency and mitigating numerical drift due to floating-point errors. This is achieved via post-integration normalization.

Across all scenarios, the model exhibits expected qualitative behaviors:

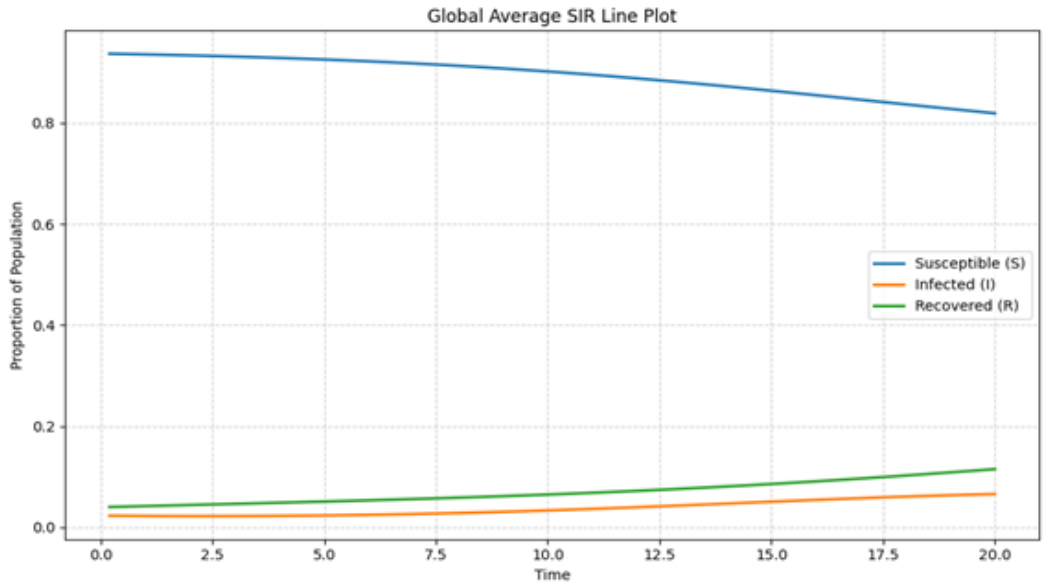
- **Higher** β: Faster and broader propagation; earlier infection peaks
- **Higher** γ: Rapid decline in infections; quicker recovery
- **Low** $\beta + \gamma$: Slower dynamics; longer persistence of initial state

These trends confirm that the simulation is sensitive to parameters and effectively captures distributed information and congestion dynamics.

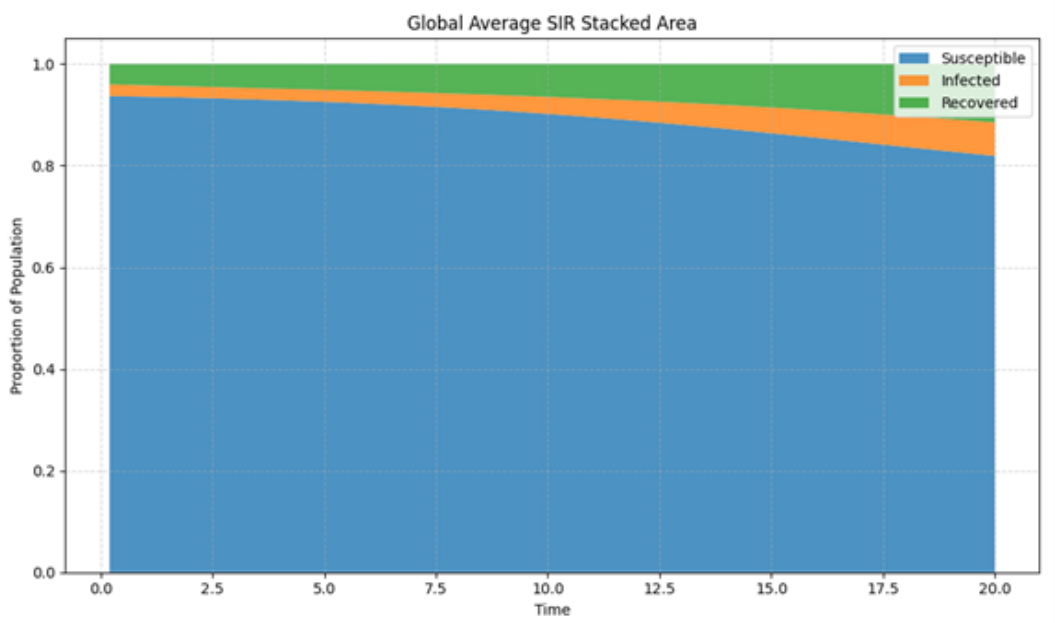
1.8.2 Visualization

Global SIR Evolution

We first provide an overview of the global trends of the SIR variables:



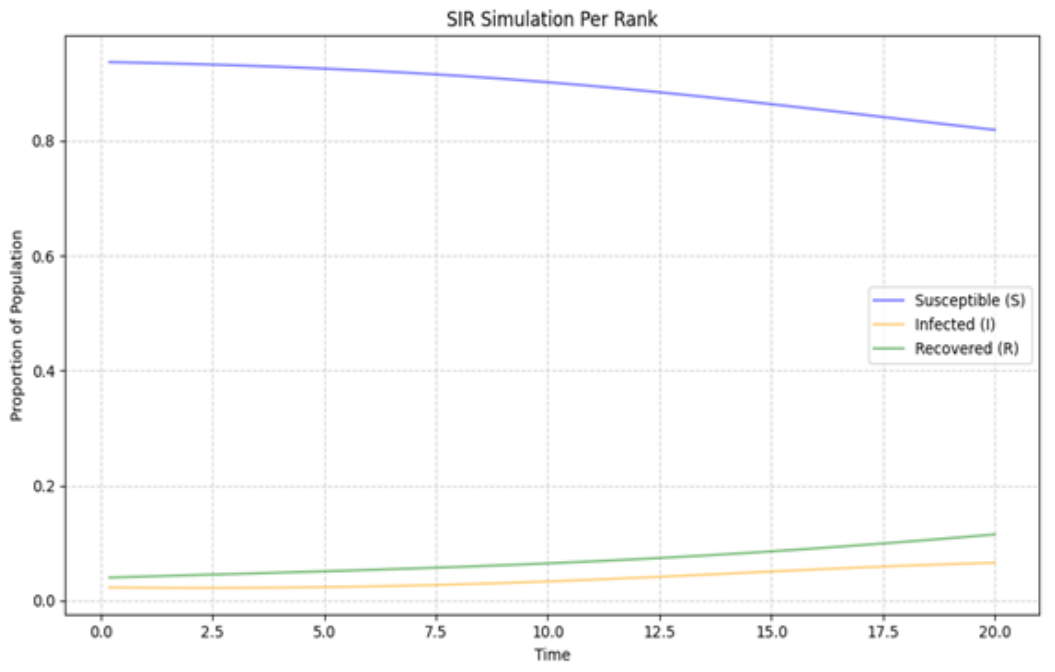
Shows the global average values of Susceptible (S), Infected (I), and Recovered (R) over time, plotted as line graphs. It confirms the expected dynamics: S steadily decreases, while I and R gradually increase.



Presents the same data in the form of a stacked area chart, emphasizing the total population conservation (S + I + R = 1) and the proportion each state occupies throughout the simulation.

Per-Rank Behavior

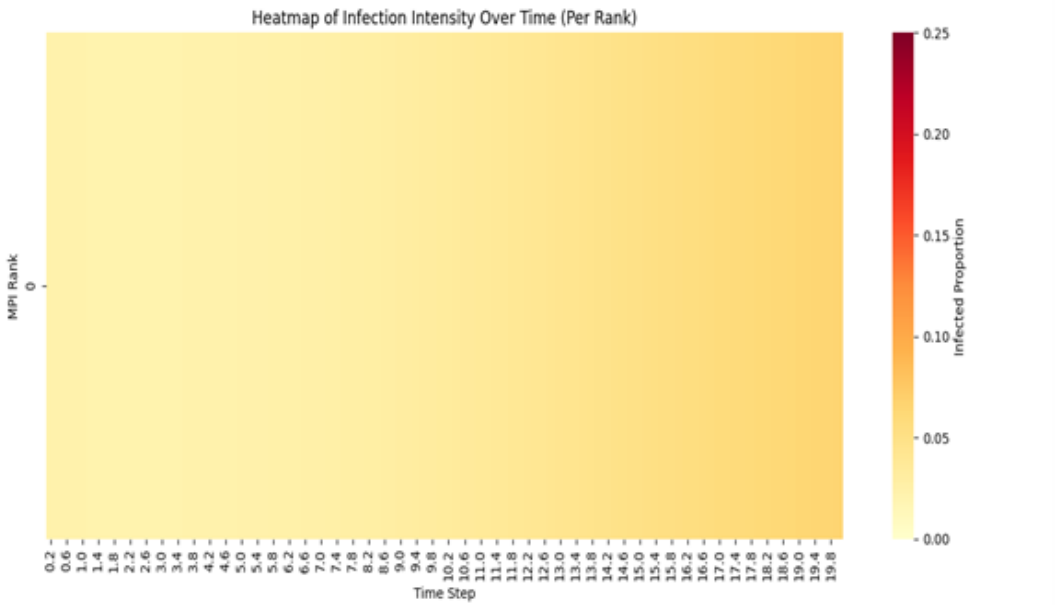
Because the simulation is distributed across MPI processes, it is important to visualize the behavior of each rank individually.



Displays the evolution of S, I, and R within each MPI rank using line plots. These trends reveal differences in infection dynamics between regions, reflecting the spatial heterogeneity.

Infection Propagation and Peak Timing

To analyze the infection wave's temporal spread and communication among ranks:



This is a heatmap that illustrates the infection intensity per rank over time. It demonstrates how the infection begins locally and gradually spreads to other ranks through the ghost cell exchange mechanism.

1.8.3 Performance and Scaling Analysis

Based on the results presented in Section 7.2, we evaluate the parallel simulation's computational efficiency. The recorded timing logs, including phase-specific breakdowns and rank-level measurements, confirm that the workload is well-balanced across MPI processes. Most of the execution time is spent in the local computation phase, while communication costs remain relatively low due to the use of non-blocking MPI operations.

The distribution of timing across ranks shows minimal variability, indicating that the domain decomposition strategy (divideIntoOptimalBlocks) effectively assigns cells to processes with balanced loads. The asynchronous communication scheme using MPI_Isend and MPI_Irecv overlaps well with computation, helping to reduce idle time during ghost cell exchange.

1.9 Conclusion and Future Work

We successfully implemented a high-performance parallel simulation framework that extends the classical SIR (Susceptible-Infected-Recovered) model to simulate smart road dynamics. The focus of this project was not on fully achieving smart road functionality, but rather on constructing a modular and scalable proof-of-concept system that demonstrates the feasibility of adapting epidemic modeling to spatially distributed vehicle networks. The model leverages 4th-order Runge-Kutta numerical integration and MPI-based parallelization with ghost cell synchronization to support large-scale simulations.

The system has shown accurate behavior, stable performance, and strong scaling properties. Timing analysis confirms efficient distribution of computational load, and visualization validates the consistency of infection dynamics across partitions.

Looking forward, several directions can further enhance the model's realism and applicability. One avenue is the incorporation of directional vehicle flow to better mimic real-world traffic behavior. Another key improvement is the use of dynamic load balancing techniques to adaptively distribute work among MPI ranks, particularly under spatially heterogeneous conditions. Additionally, the model can benefit from GPU acceleration through frameworks such as OpenACC or CUDA, which would further boost scalability. Integration with real-time sensor networks represents another valuable extension, enabling more responsive and data-driven simulations applicable to intelligent transportation systems.

This project lays a solid foundation for future research and development in parallel simulation of cyber-physical infrastructures.

Bibliography

[1] John C. Butcher. *Numerical Methods for Ordinary Differential Equations.* John Wiley & Sons, 2003 (cit. on p. 5).