TUBAF

Die Ressourcenuniversität.
Seit 1765.

FAKULTÄT FÜR MATHEMATIK UND INFORMATIK

TECHNISCHE UNIVERSITÄT BERGAKADEMIE FREIBERG

HPC Project

Disease Spread Simulation on a 2D Grid using open MPI and openMP

Author: Omkar Kolekar
Submission Date: 26 June 2024

Contents

1 Introduction

Infectious diseases persist as a formidable challenge to global health, underscoring the urgent need for advanced computational methods to study their transmission dynamics comprehensively. Addressing this need, the current simulation harnesses the robust capabilities of parallel programming frameworks, specifically OpenMPI and OpenMP, implemented within the C++ programming language. These frameworks facilitate the modeling of infectious disease spread within a simulated population by enabling efficient multi-threading and distributed computing. Through the utilization of OpenMPI and OpenMP, the simulation aims to investigate the effectiveness and scalability of parallel computing when applied to large-scale epidemic scenarios. Furthermore, a Python script is integrated into the workflow to provide visualization capabilities, generating intuitive graphical representations of the simulation results. This study is dedicated to evaluating the computational advantages and practical applications of parallel programming in the context of epidemiological simulations.

Parallel computing revolutionizes the speed and efficiency of computational tasks by concurrently utilizing multiple processors or computing resources. Unlike traditional serial computing, where tasks are processed sequentially, parallel computing allows for simultaneous execution, leading to significant performance enhancements and scalability. This approach is essential for tackling intensive computations in fields such as scientific simulations, big data analytics, AI development, and real-time processing, where speed, efficiency, and scalability are paramount for advancing technological capabilities and driving innovation.

1.1 Chapters

The report is organized into six chapters as follows.

Chapter 1: Introduction to the report.
Chapter 2: Problem statement description.
Chapter 3: Algorithm description and User manual.
Chapter 4: Validation and testing.
Chapter 5: Strong and weak scalability test.
Chapter 6: Conclusion.

2 Problem Description

In this project, a parallel simulation of disease spread on a grid is to be implement, inspired by SIR (Susceptible, Infected, Recovered) or SEIR (Susceptible, Exposed, Infected, Recovered) models. This simulation will use MPI for spatial partitioning of the grid and OpenMP for parallel processing within each grid segment.

2.1 Grid States

Each cell in the grid can have one of the following states:

- **Susceptible (Healthy)**: Can be infected.
- **Infected**: Can infect other cells and will become healthy after a certain time.
- **Recovered (Immune)**: Temporarily immune after recovering from infection.

2.2 Rules for State Changes

1. **Infection**: A healthy cell adjacent to an infected cell becomes infected with a probability p (infection probability) in the next time step.
2. **Recovery and Immunity**: An infected cell has a probability q (recovery probability) to become healthy in the next time step. After recovery, it becomes immune for t time steps. After the immunity period, the cell becomes susceptible again.

3 Algorithm and User Manual

3.1 Flow Chart

The figure 3.1 gives an overview of the working algorithm.

3.2 User Manual

The user should ensure that the Linux system has openmpi and openmp installed. Then using the following commands the program can be run.

```
module load openmpi

mpicxx -fopenmp EpidemiSimPar.cpp

export OMP_NUM_THREADS=1;

mpirun -np 3 ./a.out
```

The software is programmed in such a way that each rank will write the state of the matrix to a text file matrix_output_rank_X.txt (where: 'X' rank number) irrespective of the number of ranks. To visualize the data, it is essential to download the '.txt' files generated by the program onto a separate computer equipped with Python and matplotlib, and execute the Animate.py script. Make sure that the downloaded '.txt' files and the python script are in the same folder.
If matplotlib is not installed, the user can use the following commands to install it.

```
pip install matplotlib
```

To run the python script use the following command.

```
python3 Animate.py
```

It is important to note that the python script works only for 3 ranks and with '.txt' files.
For scalability testing, adjust the number of rows in the program to an appropriate value and disable the printing of matrices.

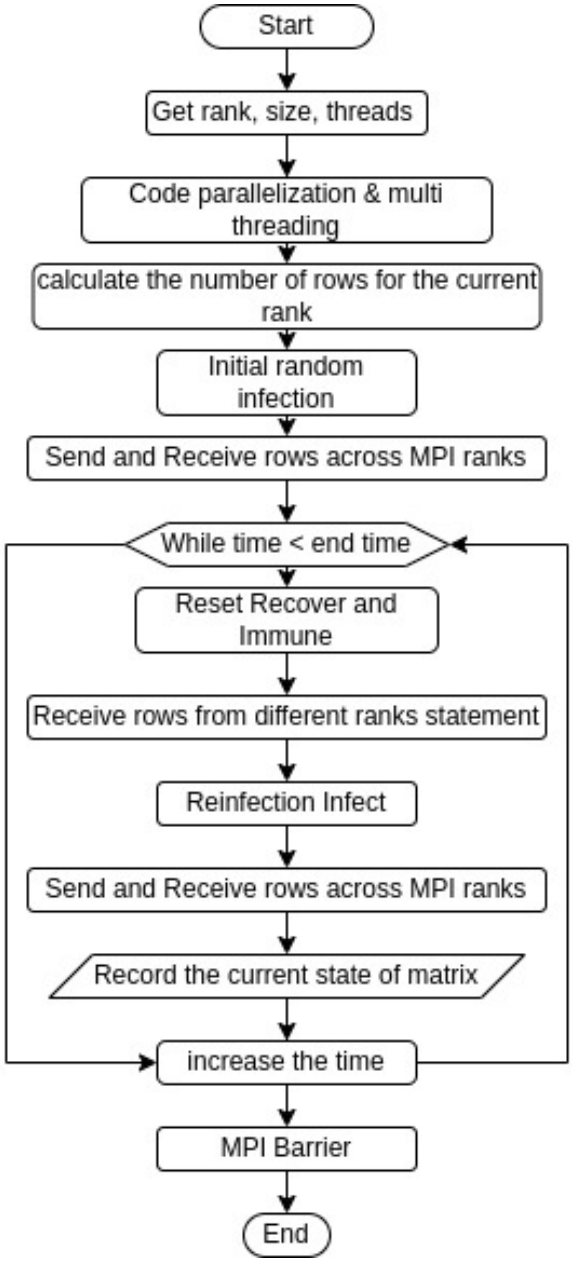
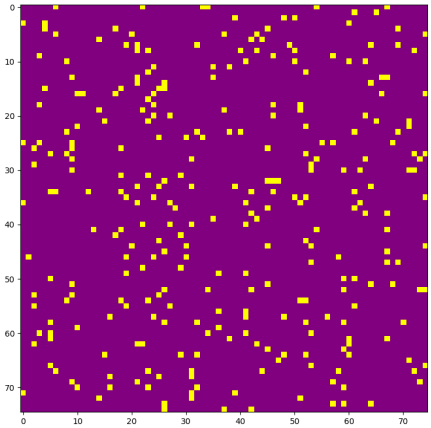


Figure 3.1: Algorithm flow chart

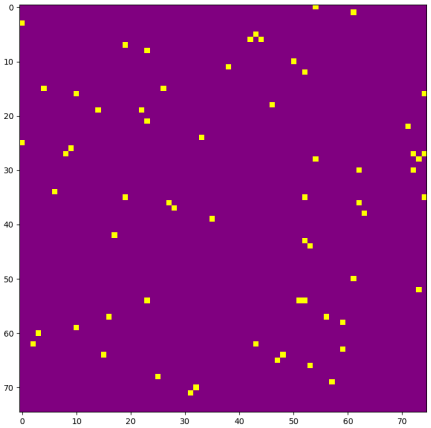
4 Validation and testing

4.1 Visualization of output

The below few pictures are snaps from first two time steps. The size of the matrix was 75x75.

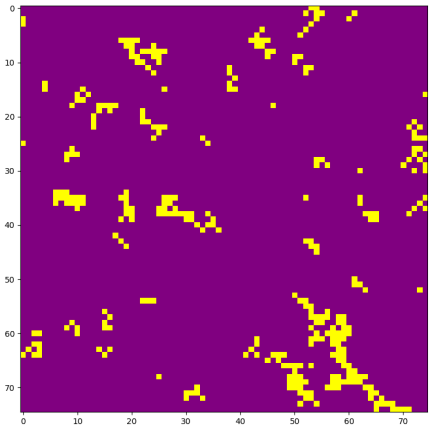


(a) Infected

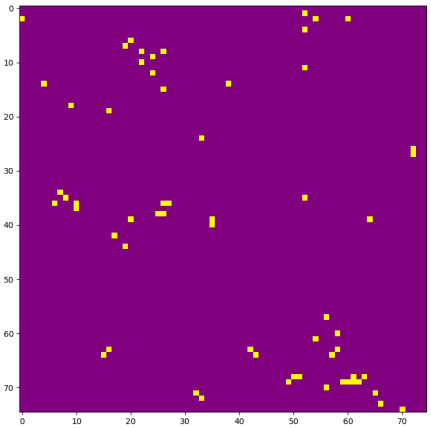


(b) Recovered

Figure 4.1: Infection state at time t = 0



(a) Infected



(b) Recovered

Figure 4.2: Infection state at time t = 1

4.2 Testing

Since I have chosen row major distribution of matrix across all the MPI ranks, the testing in MPI is based on the boundary rows of the matrices.
The software is programmed in such a way that each rank will write the state of the matrix to a text file matrix_output_rank_X.txt (where: 'X' rank number) irrespective of the number of ranks. For a new time step a new matrix will be appended under the previous matrix in the same text file.

Using the cat command in the linux bash terminal, following pictures were obtained. I have marked the important rows of the respective rank. The red rectangle represent the state of the row at time t = 0. The green rectangle represent the state of the row at time t = 1. The boundary rows show no irregularities in all the ranks, which ensures the proper implementation of the MPI parallelization.

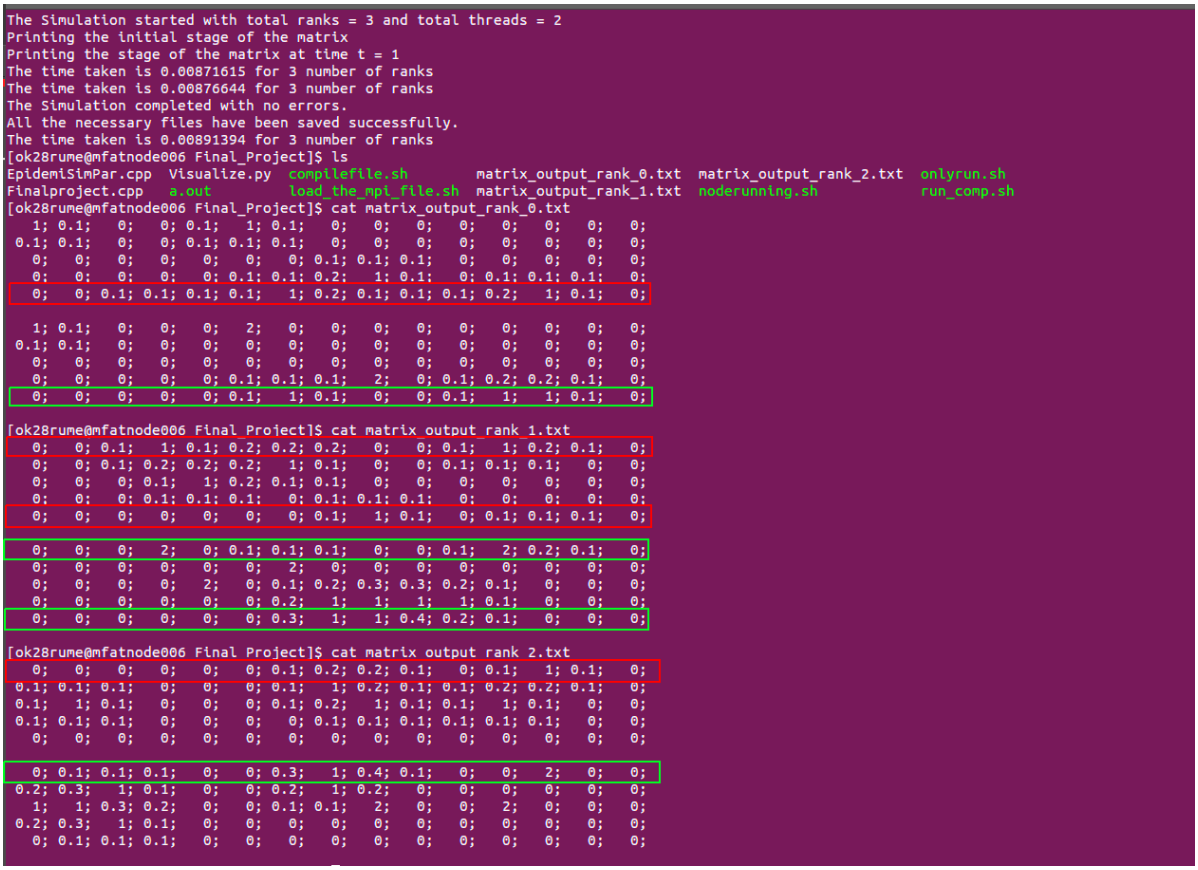


Figure 4.3: Boundary row testing result

5 Strong and Weak Scabality Test

Scaling tests are employed to assess the parallel performance of a computational code, with the objective of optimizing the performance of an application or system. These tests enable the determination of the most efficient allocation of computational resources for a given problem. It is noteworthy that during scaling tests, the grid is not printed to the output file. Adjustments to the grid size and the number of iterations are implemented through code modifications, rather than being provided as user inputs for each trial.

5.1 Strong scaling test

A strong scaling test is a test in which the problem size remains constant while the computing power is variable. The size of the grid was 2000 X 2000 and the number of time steps were 50. The number of cores (Processors) were varied from 1 to 40 and the threads per processor was also varied to observe the effect of multi-threading. The table 5.1 compares the effect of multi threading per core on reduction in computational time as the number of cores (processors) increases. The figure 5.1 is a visual representation of table 5.2.

5.2 Inference

5.2.1 Without multi threading

The figure 5.1 illustrates the reduction in computational time as the number of cores (processors) increases, each utilizing a single thread per rank.

- With increase in the number of cores (processors) the computational time does decrease up to a certain limit.
- Beyond this limit there is no overall drop in the computational time as the overhead increases

5.2.2 With multi threading

The table 5.1 illustrates the variation in computational time as the number of cores (processors) changes, each utilizing various threads per rank.

- There is an increase in computational time observed due to multi threading, which confirms the statement in lecture 'Hybrid MPI/openMP parallelization', that the 'hybrid MPI/OpenMP can be SLOWER than pure MPI'.

- This may be due to the increased overheads, thread management and increase in communication complexity.

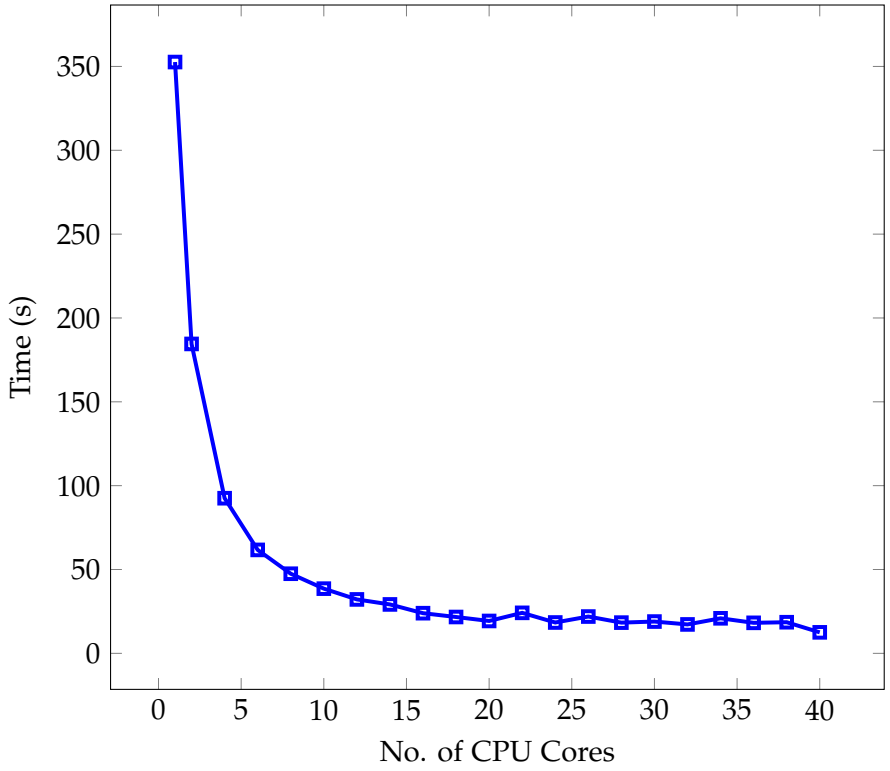


Figure 5.1: Strong Scaling Test

No. of Cores	Time (s)
1	352.621
2	184.518
4	92.5089
6	61.686
8	47.4806
10	38.5817
12	32.1541
14	29.1692
16	23.9454
18	21.6734
20	19.3101
22	24.1548
24	18.3978
26	21.9483
28	18.3226
30	18.9574
32	17.2403
34	20.8977
36	18.1985
38	18.5591
40	12.5239

Figure 5.2: Strong Scaling Test

MPI Ranks	**Threads**	**Core**	**Time**	**Time by MPI Ranks**	**Difference**
1	4	4	450.775	92.5089	358.2661
2	4	8	286.005	47.4806	238.5244
2	8	16	435.278	23.9454	411.3326
4	8	32	111.164	17.2403	93.9237
8	5	40	64.0392	12.5239	51.5153

Table 5.1: Performance metrics of MPI Ranks, Threads, and Core with corresponding times

5.3 Weak scaling test

Weak scaling tests assess how well a program can manage larger problems by increasing resources while maintaining the workload per processing unit constant. This capability is crucial as it demonstrates the program's ability to handle larger problems effectively when sufficient computational resources are available. In this test the number of cores were doubled from 1 to 32 for each run. The test is conducted for three different load cases i.e. 10E5 cells, 10E6 cells and 5*10E6 cells per core respectively. For all trials the time steps were kept constant. The graph 5.3 was obtained from the data in the table 5.4 which summarises the wall time required to complete the computation under different load cases.

5.3.1 Inference

- The curve with 10E5 cells per core exhibits an almost flat slope, which is indicative of an ideal scenario for the weak scaling test. However, this is not the ideal load to measure the weak scaling test as the wall time is below 50 s.
- The curve with 10E6 cells per core is almost flat as well which is an ideal behaviour.
- The slope of the last curve with 5*10E6 cells per core deviate significantly from the ideal behaviour. The curve has a positive slope.

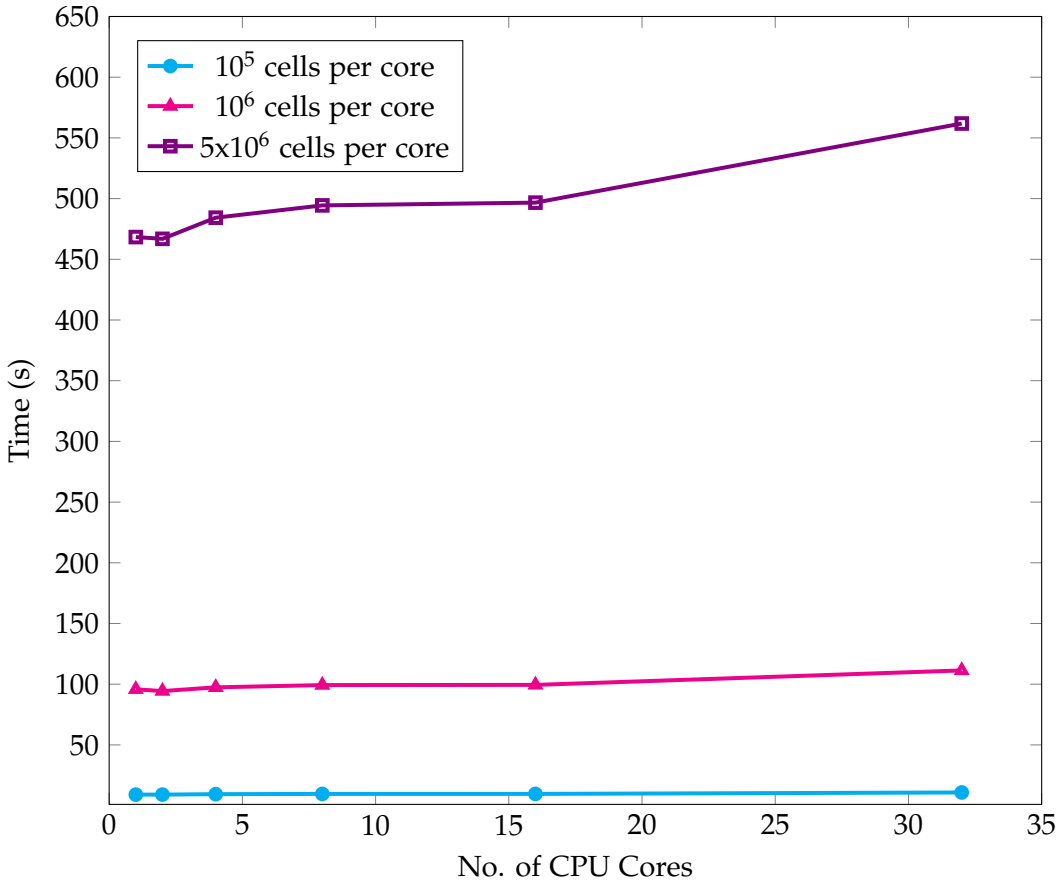


Figure 5.3: Weak Scaling Test

Number of cores	Average wall time (s)		
	10E5 cells	10E6 cells	5*10E6 cells
1	9.0179	95.8067	468.302
2	9.061	94.3319	466.859
4	9.41807	97.4106	484.273
8	9.62055	99.27	494.4
16	9.63562	99.4188	496.673
32	10.8104	111.341	561.838

Figure 5.4: Weak Scaling Test Data

6 Conclusion

In this project, the spread of disease within a community was simulated. The simulation was programmed in C++ utilizing MPI and OpenMP. The program outputs a matrix in the form of text files, with the number of text files generated corresponding to the number of threads used. Race conditions within the threads were avoided through the proper placement of barriers. Strong scaling tests yielded good results, demonstrating a significant reduction in computation time with parallelization. However, the effect of multi-threading was found to be counter intuitive, likely due to the increase in overhead. Weak scaling tests indicated that the program performs well for approximately 10E6 cells per core. In conclusion, the simulation effectively leverages parallel computing to model the spread of disease, significantly improving computational efficiency.

List of Figures

List of Tables