Title: Parallelization of Disease Propagation Algorithm (DPA)

Sungsoo Kim (EID: sk38224)

Abstract

The advent of COVID-19 novel virus from Wuhan, China and its spread across the world is making devastating humanitarian and economic impact. With its ever-increasing growth rate, it poses a large challenge to healthcare officials and others to gauge its effect and deploy countermeasures to control its spread. Such situation has inspired me to develop a disease propagation modeling algorithm to provide a better understanding of the COVID-19 virus. This algorithm will be based on C++ and OpenMP to maximize performance.

The project is divided into two major components: 1. Creating a general disease propagation model and 2. Parallelizing the code for optimal computation. The algorithm will be based on the SIR (Susceptible, Infectious and Recovered) model and other specific details of these sections will be determined through additional research. This research phase will help identify other major variables to consider and further enhance the accuracy of the simulator. Moreover, reference to existing mathematical models of disease propagation will also provide a general guidance in the development of the serial code.

Introduction

The simulation of the disease propagation model is developed based two major classes: Population and Person Additionally, there are system variables that govern the simulation. First, the system variables are set as the following:

System Variables

Field Name	Description	Value
INFECTION_RATE	Possibility of Infection when in contact	0.50
INTERACTION_NUM	Mean, S.D. of Interactions (per day)	Mean: 12, S.D: 3
DORMANT_PERIOD	Average Dormant Period of Virus	14 day
RECOVERY_PERIOD	Mean, S.D. of Recovery	Mean: 14, S.D.: 2
CHANCES	Random Chance of Infection	0 < CHANCES < 1 (Random)

Person Class

Field Name	Description	Value
Status	Current Status of the Patient	-
- Infected	The individual will be assumed not in self-isolation up to day 14. Once discovered, the individual will be recovering based on the recoveryDate field.	Status >= 1
- Recovered	Individual has recovered from the virus and gained immunity	Status = -1
- Susceptible	Individual has no immunity and is susceptible to the virus	Status = 0
- Inoculated	Individual has immunity to the virus	Status = -2
Interaction	Interaction count calculated based on national normal distribution. The numbers are renewed every day for accurate simulation	Individual Normal Distribution values based on the INTERACTION_NUM field
recoveryDate	Recovery date calculated based on the normal distribution for COVID-19. The value is initialized with the People Vector.	Individual Normal Distribution values based on the INTERACTION_NUM field

Population Class

Field Name	Description	Value
Size	Number of People in the Community	10,000,000
People <Vector>	Vector of Person objects with a size of "Size"	<Vector>
Day	Current Day of the Simulation	0 to user defined date

Output File

Name	Format
Disease_Simualation.csv	Day || Total Sickness || Uninfected || Recovered & Inoculated

To optimize the simulation for the purpose of this project to parallelize, there were several hypotheses taken.

1. The modeling is based on a single community of people, and there are no external interactions with any interaction with the external communities.

2. Infected individuals will continue the social interaction during the virus dormant period (14 days for COVID-19) and then progress into the self-isolation period when he/she will begin the recovery.

3. There is no mutation to the virus – Once the individual has recovered from the virus, the individual is then immune
4. The chances of getting infected is kept constant throughout the simulation (Regardless of age, gender and other possible factors)
5. No external social act limitations are in place – No social distancing & Isolation except for the infected individuals with signs of infection.
6. Out of the interactions, we assume that 10% of them are “new” interactions whereas the rest are composed of our close friends, including our family, neighbors and others that we interact on a daily / bi-daily basis.

In terms of the variables, the population was set to 4,000,000 people to simulate relatively large sized city, the recovery and the dormant periods were taken from the WHO’s latest report on COVID-19 (April, 20’). Additionally, the chances of infections during an interaction with a virus holder was randomly selected to 30% and the number of interactions per day was taken from a study (Martire et al., 2018).

Project Objectives

1. *Code Development*
   A. Create a Disease Propagation Algorithm based on the SIR model
   B. Create a scalable solution that can vary the scope of the simulation
      i. Introduce Objects *E.g. Nations, states etc.*
   C. Compare output values to known data (*Johns Hopkins Coronavirus Research Center*)
2. *OpenMP Implementation*
   A. Implement OpenMP functions to enable parallelization of the code
   B. Generate a benchmark report to optimize parallel performance of the code
   C.

Implementation

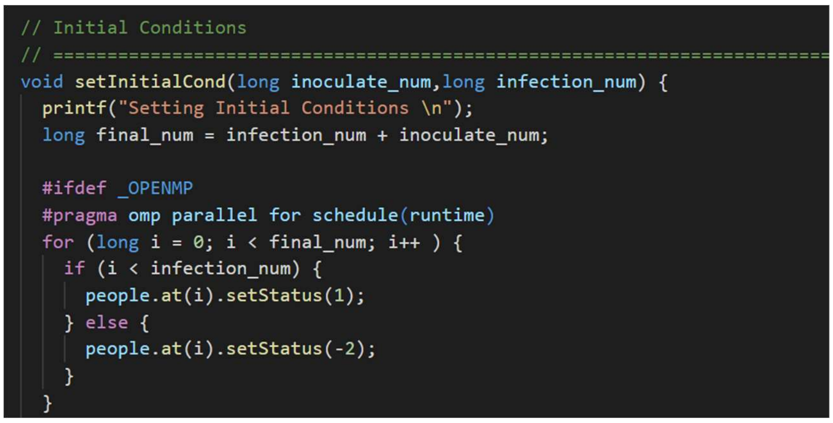
The simulation process can be divided into four different parts: Initialization of the People vector, setting initial conditions, virus simulation, and Counting. Here, the report will go through each part in detail and discuss how the program was converted from a serial to a parallel code.

1. *Initialization of the People Vector*

The program holds the information of a single person in a vector. A vector is a dynamic datatype and the initialization of the People vector took relatively less compared to the other hefty processes taken in the program. Therefore, the parallelization for this section was not done. However, in the case that the program is extended to be used for multiple communities (using multiple Populations class), it is possible to place a OpenMP for loop to iterate through different population classes to reduce the computation time.

2. *Setting Initial Conditions*

Once the People vector is constructed, the program takes in the initial conditions set by its user. The user specifies the number of infected people, number of inoculated people. Initially, the setup of the two conditions were done in two for loops. However, to optimize it for OpenMP application, the two for loops were combined.



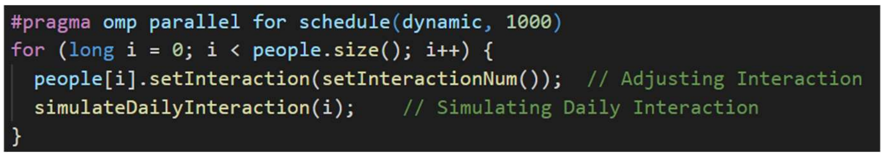
```
// Initial Conditions
// =========================================================================
void setInitialCond(long inoculate_num,long infection_num) {
  printf("Setting Initial Conditions \n");
  long final_num = infection_num + inoculate_num;

  #ifdef _OPENMP
  #pragma omp parallel for schedule(runtime)
  for (long i = 0; i < final_num; i++ ) {
    if (i < infection_num) {
      people.at(i).setStatus(1);
    } else {
      people.at(i).setStatus(-2);
    }
  }
```

The initialization function takes in the two variables and goes through the for loop as the above. Because the two inputs are independent from each other, we can parallelize the loop without any race conditions. In the future, the program would be more effective if the initial conditions are set with the initialization stage.

3. *Virus Simulation*

The virus simulation is the section of the program that benefits the most from parallelization. This section divides the iteration of people in the population class and simulate human interaction person by person. Based on the variables and hypothesis outlined above, the simulate function runs the simulateDailyInteraction() function to go through an individual's interpersonal activity for the day.

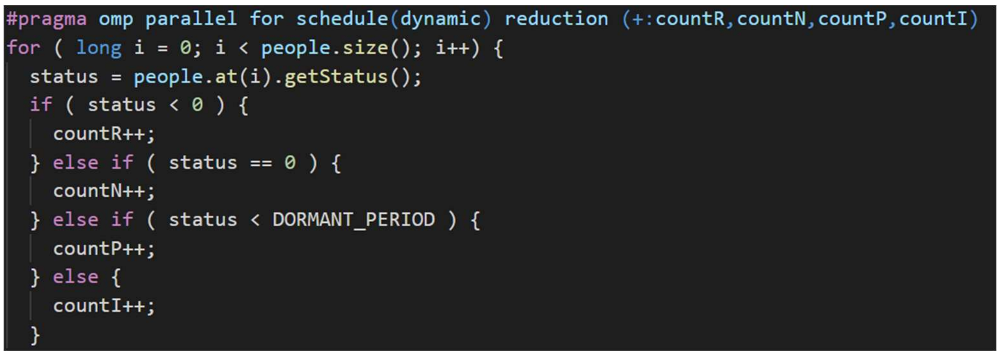


```
#pragma omp parallel for schedule(dynamic, 1000)
for (long i = 0; i < people.size(); i++) {
  people[i].setInteraction(setInteractionNum());  // Adjusting Interaction
  simulateDailyInteraction(i);    // Simulating Daily Interaction
}
```

As can be seen above, the OMP for loop is implemented to parallelize the operation. The index of the for loop indicate each person in the population. For this operation, there are chances of having a race condition. However, because most of the interactions are designed to be done to the neighboring people in the memory, a dynamic scheduler is used. By using this method, the program will have less chance of having a race condition.

4. Counting Population

The last section of the program that implements OpenMP is the counting section. Here the program goes through the Population Vector and counts the status as it goes through each person. The implementation of OMP for loop was done as below.



```
#pragma omp parallel for schedule(dynamic) reduction (+:countR,countN,countP,countI)
for ( long i = 0; i < people.size(); i++) {
  status = people.at(i).getStatus();
  if ( status < 0 ) {
    countR++;
  } else if ( status == 0 ) {
    countN++;
  } else if ( status < DORMANT_PERIOD ) {
    countP++;
  } else {
    countI++;
  }
```

Unlike other for loops in the program, the counters in each thread need to be summed by the end of the execution. Hence, this for loop introduces the reduction for all the counters. By having this, all the counters in each thread can be added up to provide a total status of the population on a given date.

5. Others

Throughout the program, random number generation functions are used to calculate the possibility of infection and to find random encounters. For the original version of the code, it was implementing the rand() function, which was not thread safe and interfered with the performance of the parallelized program. As a result, the execution time became significantly larger than the serial execution time. Hence, these random functions were replaced by the rand_r function, which takes the seed from each thread to ensure that the threads are running the random functions independently and not conflict with each other. However, due to the way rand_r() functions are set, it tended to create the same values throughout its execution so it will require some tuning for the parallelization version to be used yet it would not influence the results for this experiment.

Experiment

The main objective of the experiment is to investigate the speed up of the program with different number of threads. Here are the general outlines of the controlled variables.

Independent Variable	**Specifications**
Number of Threads	1(Serial & Parallel), 2, 4, 8, 16, 32, 64, 128, 256

Dependent Variable	**Specifications**
Execution time	Time taken for Initial Conditions & Simulation

Controlled Variables	Specifications
Population	4,000,000 People
Number of Inoculated People	1000 People
Initial Infected Individuals	2 People
Simulation Length	35 days
Hardware	TACC Stampede2 skx-dev

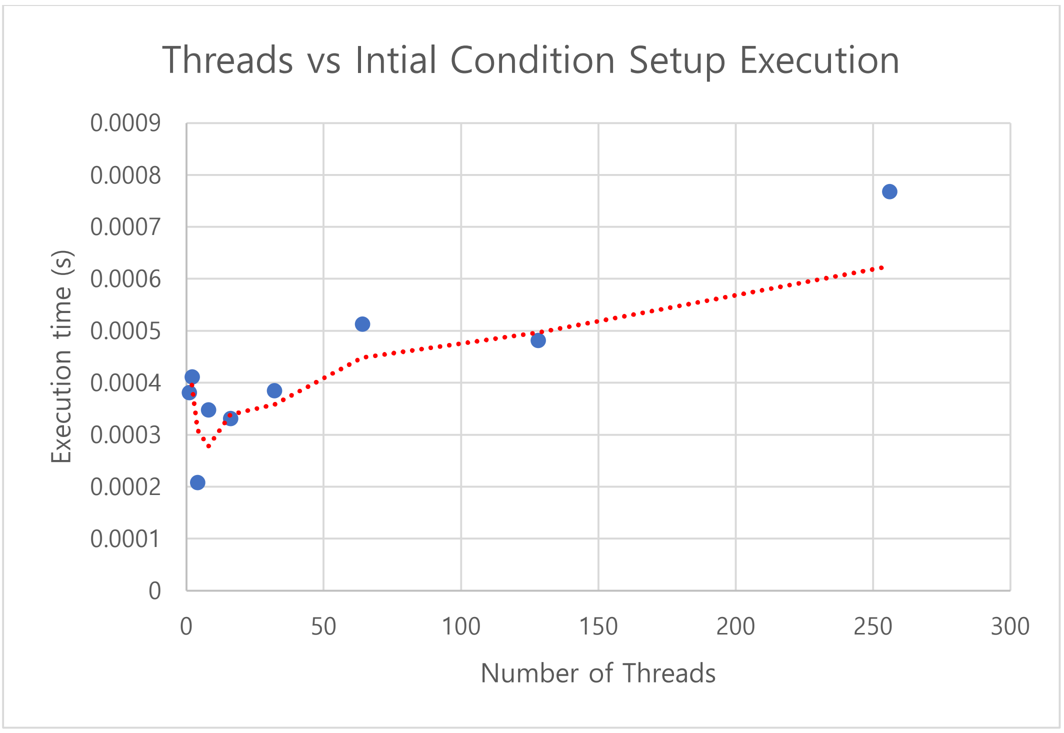
Results

1. Serial vs. Parallel (1 Thread Execution)

	Serial Execution	Parallel Execution
Initial Conditions (s)	0.000398	0.000381
Simulation (s)	259.366	259.636

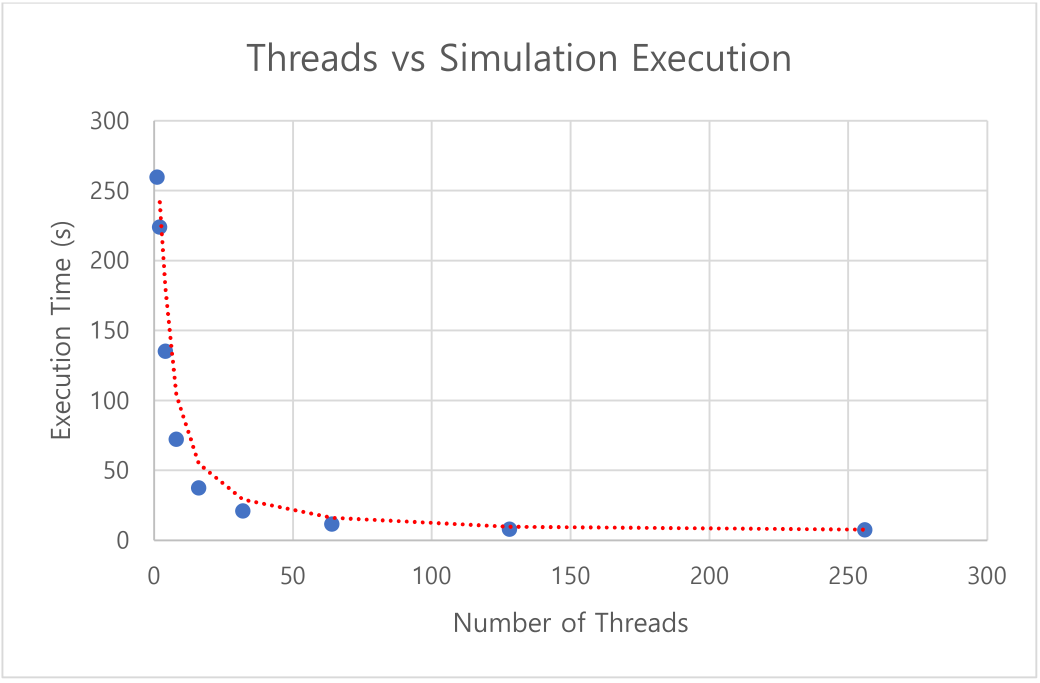
In the case of the single thread execution, the serial code and the parallelized code came out to have similar performance results. For the initial condition setup, the parallel execution resulted with better performance, where as the serial execution of the code came to be slightly faster in the disease propagation simulation.

2. Parallel Performance in Initial Conditions

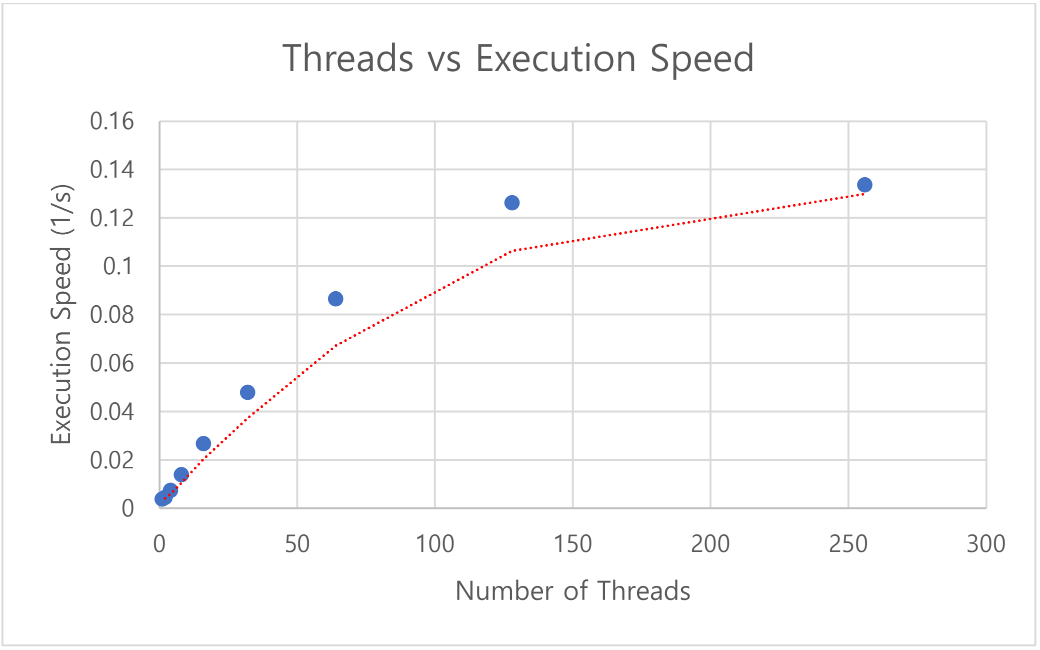


Interestingly, the time taken for initial conditions started increasing after 4 threads. While this can be an odd phenomenon to observe, we can deduce that the time required to process the section increases with more threads because there isn't enough tasks for the threads to process, making it rather inefficient to have more threads and to coordinate than having a few threads with less need to allocate tasks.

3. Parallel Performance in Simulation



Here we can see that the number of threads is inversely proportionate with the simulation execution time. Additionally, there were noticeable changes in the first few stages of the increments whereas there was relatively less impact on the execution speed as the experiment increased the number of threads.



If we take the inverse of the execution time, the relationship between the execution speed of the parallel program and the number of threads can be found. From this graph, it seems that there is a logarithmic relationship between the two variables. Yet, it evidently illustrates that the execution speed continues to rise as the number of threads increases.

Summary

As expected, the program displayed an inversely proportionate relationship for the simulation section of the software. On the other hand, the initial condition function showed that it actually becomes less effective to use more threads if the work shared is not big enough. Furthermore, during the conversion of the program, there were issues with the randomized number system where it could not generate truly unique set of random numbers. Hence, as of now, only the output generated from the serial version of the software creates can be considered useful. However, if this issue is resolved, the program will be able to handle even bigger populations and potentially start simulating interactions between different communities as well.

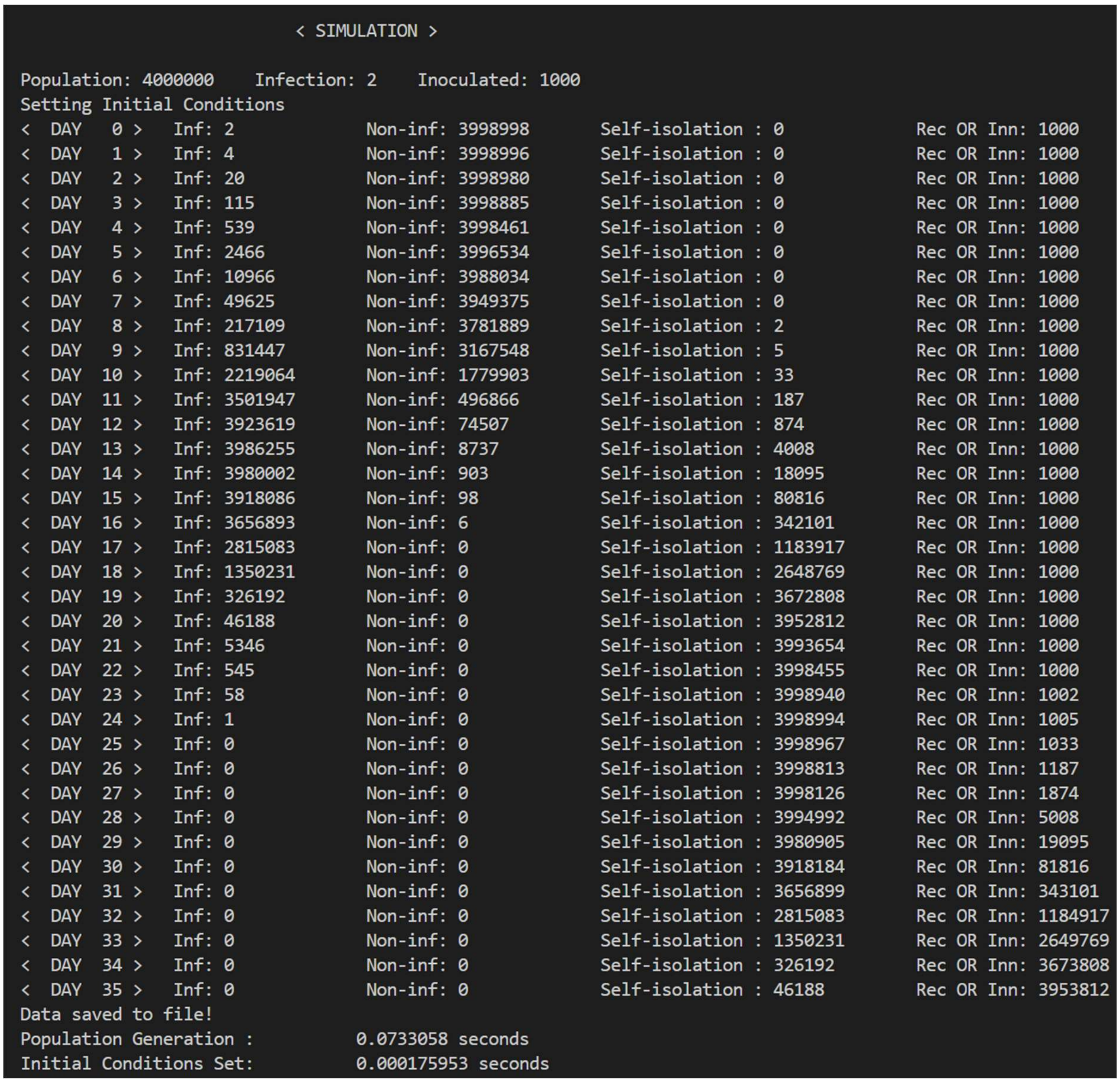
In terms of improvements, the program can combine the initialization of the Population vector with the initial condition setter to reduce the computational cost. To do this, the program will need to pre-allocated the memory for the vector. This way, each thread can independently access different elements of the vector and the program can fully benefit from parallelization.

References

Zhaoyang, R., Sliwinski, M., Martire, L., Smyth, J., & Zhaoyang, R. (2018). Age differences in adults' daily social interactions: An ecological momentary assessment study. *Psychology and Aging*, *33*(4), 607–618. https://doi.org/10.1037/pag0000242

Appendix

Program Output



```
                           < SIMULATION >

Population: 4000000    Infection: 2    Inoculated: 1000
Setting Initial Conditions
<  DAY   0 >  Inf: 2            Non-inf: 3998998     Self-isolation : 0           Rec OR Inn: 1000
<  DAY   1 >  Inf: 4            Non-inf: 3998996     Self-isolation : 0           Rec OR Inn: 1000
<  DAY   2 >  Inf: 20           Non-inf: 3998980     Self-isolation : 0           Rec OR Inn: 1000
<  DAY   3 >  Inf: 115          Non-inf: 3998885     Self-isolation : 0           Rec OR Inn: 1000
<  DAY   4 >  Inf: 539          Non-inf: 3998461     Self-isolation : 0           Rec OR Inn: 1000
<  DAY   5 >  Inf: 2466         Non-inf: 3996534     Self-isolation : 0           Rec OR Inn: 1000
<  DAY   6 >  Inf: 10966        Non-inf: 3988034     Self-isolation : 0           Rec OR Inn: 1000
<  DAY   7 >  Inf: 49625        Non-inf: 3949375     Self-isolation : 0           Rec OR Inn: 1000
<  DAY   8 >  Inf: 217109       Non-inf: 3781889     Self-isolation : 2           Rec OR Inn: 1000
<  DAY   9 >  Inf: 831447       Non-inf: 3167548     Self-isolation : 5           Rec OR Inn: 1000
<  DAY  10 >  Inf: 2219064      Non-inf: 1779903     Self-isolation : 33          Rec OR Inn: 1000
<  DAY  11 >  Inf: 3501947      Non-inf: 496866      Self-isolation : 187         Rec OR Inn: 1000
<  DAY  12 >  Inf: 3923619      Non-inf: 74507       Self-isolation : 874         Rec OR Inn: 1000
<  DAY  13 >  Inf: 3986255      Non-inf: 8737        Self-isolation : 4008        Rec OR Inn: 1000
<  DAY  14 >  Inf: 3980002      Non-inf: 903         Self-isolation : 18095       Rec OR Inn: 1000
<  DAY  15 >  Inf: 3918086      Non-inf: 98          Self-isolation : 80816       Rec OR Inn: 1000
<  DAY  16 >  Inf: 3656893      Non-inf: 6           Self-isolation : 342101      Rec OR Inn: 1000
<  DAY  17 >  Inf: 2815083      Non-inf: 0           Self-isolation : 1183917     Rec OR Inn: 1000
<  DAY  18 >  Inf: 1350231      Non-inf: 0           Self-isolation : 2648769     Rec OR Inn: 1000
<  DAY  19 >  Inf: 326192       Non-inf: 0           Self-isolation : 3672808     Rec OR Inn: 1000
<  DAY  20 >  Inf: 46188        Non-inf: 0           Self-isolation : 3952812     Rec OR Inn: 1000
<  DAY  21 >  Inf: 5346         Non-inf: 0           Self-isolation : 3993654     Rec OR Inn: 1000
<  DAY  22 >  Inf: 545          Non-inf: 0           Self-isolation : 3998455     Rec OR Inn: 1000
<  DAY  23 >  Inf: 58           Non-inf: 0           Self-isolation : 3998940     Rec OR Inn: 1002
<  DAY  24 >  Inf: 1            Non-inf: 0           Self-isolation : 3998994     Rec OR Inn: 1005
<  DAY  25 >  Inf: 0            Non-inf: 0           Self-isolation : 3998967     Rec OR Inn: 1033
<  DAY  26 >  Inf: 0            Non-inf: 0           Self-isolation : 3998813     Rec OR Inn: 1187
<  DAY  27 >  Inf: 0            Non-inf: 0           Self-isolation : 3998126     Rec OR Inn: 1874
<  DAY  28 >  Inf: 0            Non-inf: 0           Self-isolation : 3994992     Rec OR Inn: 5008
<  DAY  29 >  Inf: 0            Non-inf: 0           Self-isolation : 3980905     Rec OR Inn: 19095
<  DAY  30 >  Inf: 0            Non-inf: 0           Self-isolation : 3918184     Rec OR Inn: 81816
<  DAY  31 >  Inf: 0            Non-inf: 0           Self-isolation : 3656899     Rec OR Inn: 343101
<  DAY  32 >  Inf: 0            Non-inf: 0           Self-isolation : 2815083     Rec OR Inn: 1184917
<  DAY  33 >  Inf: 0            Non-inf: 0           Self-isolation : 1350231     Rec OR Inn: 2649769
<  DAY  34 >  Inf: 0            Non-inf: 0           Self-isolation : 326192      Rec OR Inn: 3673808
<  DAY  35 >  Inf: 0            Non-inf: 0           Self-isolation : 46188       Rec OR Inn: 3953812
Data saved to file!
Population Generation :         0.0733058 seconds
Initial Conditions Set:         0.000175953 seconds
```