EECS587 Final Project: Plague-Spreading Simulation using OpenMP

Wei-Cheng Chiang | Alan Zeng
imarthur@umich.edu | zizien@umich.edu

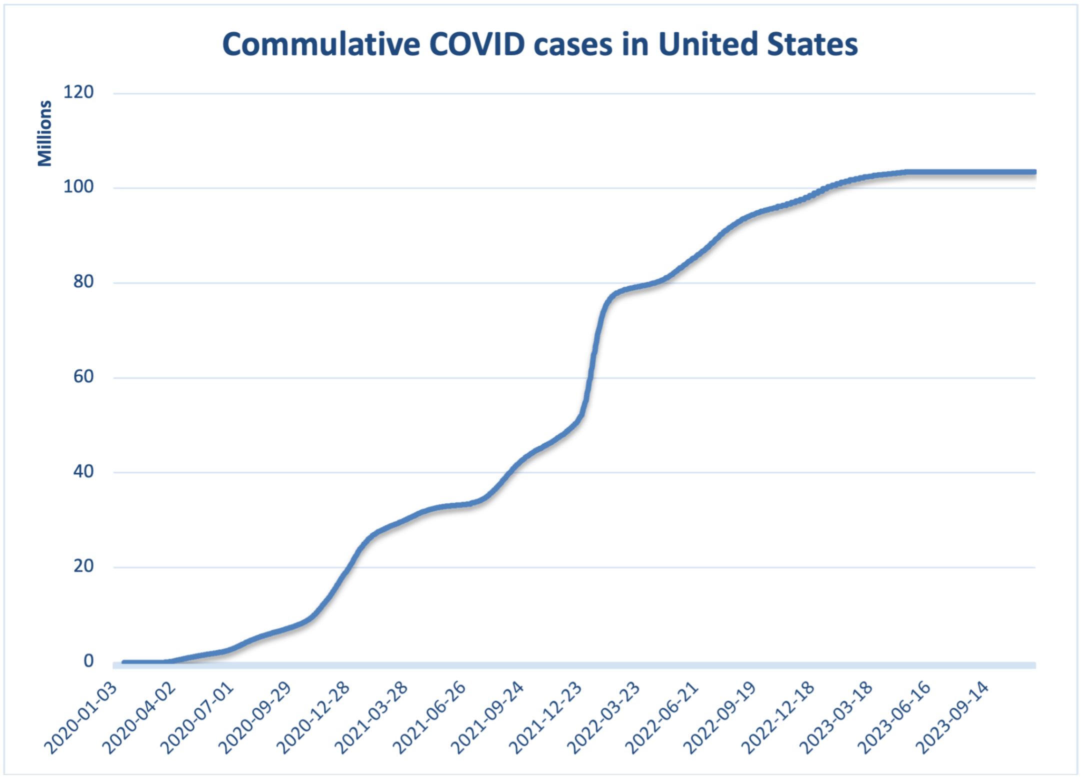
1. Research problem

Our research addressed the dynamics of COVID-19 spread by integrating a spreading plague model with a group movement model, to simulate real-world scenarios by adding some details from existing observations. The subsequent model allowed us to capture the transmission of the virus, taking into account factors such as population density and transmission probabilities. Concurrently, the group movement model provided a dynamic representation of how individuals move within a population, considering factors like reluctance to travel long distances and tendencies to travel to regions of lower density of population. This comprehensive approach aimed to mimic real-world complexities, enabling a more accurate representation of the virus's spread within a population. By combining these models, we looked into the impact of various factors on the transmission dynamics of COVID-19, considering both the biological aspects of the virus and the behavioral aspects of human movement.

2. Preliminary study

2.1 Infection Rate

According to survey data by Mathieu et al. that gives daily updates on COVID-19 case reports worldwide, we can look at the growth rate of the cases in great detail.



Reference: Edouard Mathieu, Hannah Ritchie, Lucas Rodés-Guirao, Cameron Appel, Charlie Giattino, Joe Hasell, Bobbie Macdonald, Saloni Dattani, Diana Beltekian, Esteban Ortiz-Ospina and Max Roser (2020) - "Coronavirus Pandemic (COVID-19)". Published online at OurWorldInData.org. Retrieved from: 'https://ourworldindata.org/coronavirus' [Online Resource]

Based on our analysis of data spanning from 2020 to 2022, the average daily infection rate for COVID-19 within this timeframe is calculated to be 0.24%. Employing an epoch duration of 50 days, we determined an equivalent infection rate of 12.7%. Consequently, we have set the infection rate parameter in our code to 0.13 to accurately represent the observed patterns in disease transmission over this period.

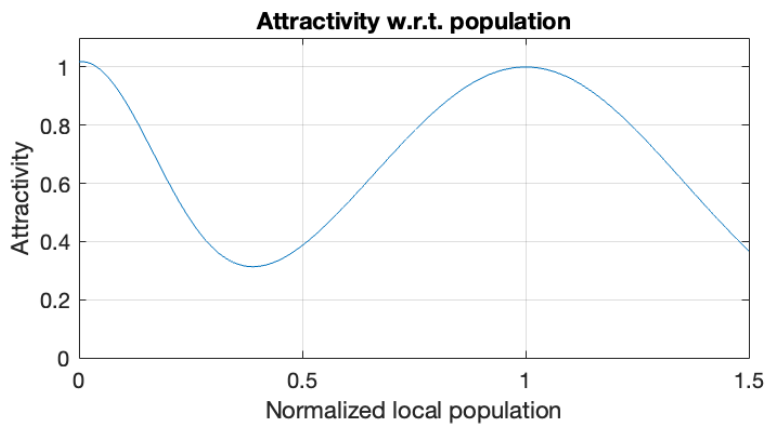
2.2 Flock movement model

Based on the research by Kattas et al., the movement of a group of pigeons is governed by a set of non-linear ODEs (Dieck Kattas et al., 2012). The study incorporates a fixed number of nearest neighbors as a neighborhood strategy for interactions between individuals, based on prior analyses indicating its relevance in bird flocking dynamics. Adopting this concept, we look at the neighborhood of each location point and determine the movement of each personnel based on two factors, the population and the distance to travel to move to that location. The probability is calculated as follows. Let's say the location i has n neighbors, labeled $j, j = 1,2,...,n$. At each possible destination, denote the distance between locations i and j to be d_{ij}, and the current population at location j to be M_j. The probability for personnel to travel from location i to location j is computed to be:

$$P_{ij} = P(x^{n+1} = j \,|\, x^n = i) = \frac{W_{ij}}{\Sigma^n{}_{k=1} W_{ik}}$$

$$W_{ij} = W_{pop}(M_j) \,/\, d_{ij}$$

Here we proposed an attractivity function $W_{pop}(M_j)$ to mimic the moving tendency of people. The function value is plotted in the figure below:



The attractivity function is proposed to mimic the overall trend of people's migration, where people may tend to avoid highly populated regions and move to regions with fewer people. However, it is also a possibility that people tend to follow the movement of a group. Thus, we added a peak near the moderately populated scenario as a local attractor for people.

3. Parallelism

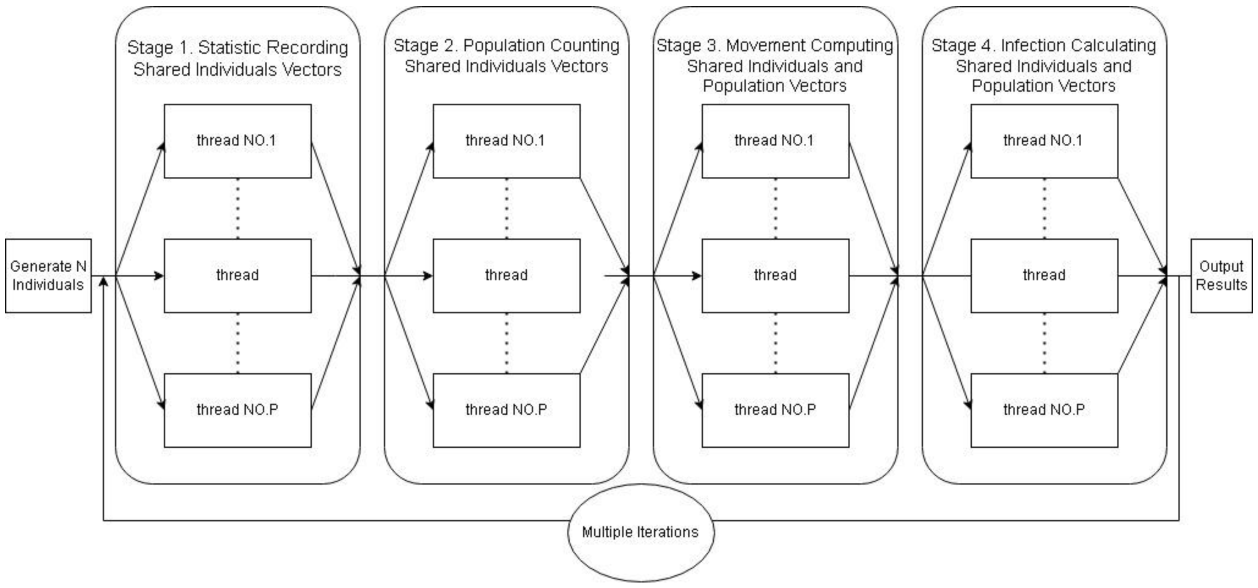
The challenges parallelizing this problem come in two parts. First, the population density is dynamically changing with the movement of people, and the simulation should be able to keep track of that change correctly and consistently despite that the dynamics of each individual or each location may be treated in parallel. Second, the distance between each personnel to each other is also a variable to consider, and this is the most difficult part to deal with.

To calculate the population at each location, our first try was to declare a population vector having the same size N as the number of locations and make it accessible to all threads, then update the population within a parallel for loop that iterates through all individuals. However, we soon realized that it is very inefficient to do this because this approach demands extensive use of blocking and locks, as multiple individuals from different threads might all want to access the same location indices, slowing the program down. Therefore, we turn to the vector reduction feature available in the “pragma omp” header. By declaring one more local vector for each thread to record the population distribution on each chunk of individuals, and adding the populations all up after going through the people in one chunk, we end up with the correct result without meaningless waiting.

To deal with the second problem, we decided not to record the precise position of each individual in the domain. Instead, we chose to embrace one data structure – a weighted undirected graph. The use of graph structure in this work facilitates the computation and parallelism significantly. In a non-directional graph structure, only the connectivity between nodes is stored, and no spatial information is recorded. However we need spatial information to keep the whole thing going, and this is why we included the calculation for distances among nodes and derived distance values as weight features during our data preprocessing. There are two important things to note about this. First, with this setting where we strictly associate one personnel to one node in a graph, we essentially assumed a rather homogeneous distribution at each location. This point of view can be justified if the domain is sufficiently finely divided. The other assumption is that the effect of population density on the movement of each personnel is confined at a local scale, and there is no non-local effect coming into play. That is, the movement we set only takes the population from locations nearby as input, and the length scale of global or a larger regional distribution is not taken into account.

To summarize, the parallelism in our code deals with complex interactions between nodes. To simplify the question a little, we employed the graph data structure, so that we are exempt from a lot of headaches. The parallel part of our code takes care of the behavior of people moving around on the ground, and the main idea is how to acquire information from the neighbors of each location efficiently.

4. Code Structure


Generate N Individuals
Stage 1. Statistic Recording Shared Individuals Vectors
thread NO.1
thread
thread NO.P
Stage 2. Population Counting Shared Individuals Vectors
thread NO.1
thread
thread NO.P
Stage 3. Movement Computing Shared Individuals and Population Vectors
thread NO.1
thread
thread NO.P
Stage 4. Infection Calculating Shared Individuals and Population Vectors
thread NO.1
thread
thread NO.P
Output Results
Multiple Iterations

Our code executes simulations both in serial and parallel implementations with various thread counts using OpenMP. Since the computation for population and movement requires frequent inquiry for data from static shared information, this led to the choice of OpenMP for the convenience it provides with the shared memory architecture. On the other hand, there are random steps when initializing N individuals and deciding individuals' movement, to mitigate the effect of random factors, we repetitively run the same program with fixed conditions and use the average time of multiple runs as our results.

Per the parallelized version of the simulation, we used OpenMP to exploit multiple threads for concurrent processing. Each thread is responsible for simulating the behavior of a chunk of individuals. The parallel computation is broken down into four parts. First of all, record and output statistics of current epoch status, second, calculate the population at each location, third, move each individual based on the probability. Then in the last part the people will be randomly infected based on the infection rate.

In the first stage, we use OpenMP *reduction* clauses to sum three statistical numbers – current infected population, cumulative infected population, and healthy individuals. As expected, the cumulative infected count shows a similar trend to the database we referred to.

In the second stage where we determine the population at each location within the graph, each thread handles a chunk of individuals, and the *population* vector is updated for the corresponding location. To accelerate computation without using locks and avoid potential race conditions, we use OpenMP *reduction* clauses on vectors. However, default sum reduction in OpenMP doesn't support reductions on containers like vectors, maps, or sets but only variables, we use the OpenMP *declare* clause to define operator overloading on vector sum in our code. In this way, all threads would hold their own copies of *sub_population* vectors and merge into one *population* vector by the end of parallelization.

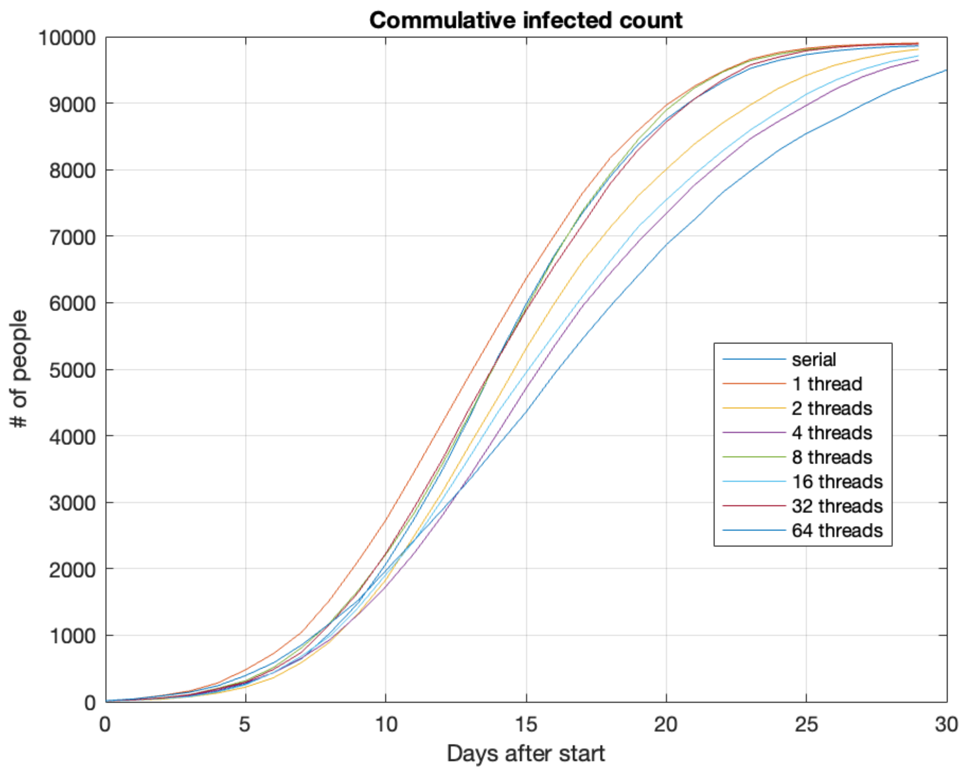
Moving on to the third step, we take care of the random movement of individuals in each location. Using the *population* data computed in the previous section, the data sharing is facilitated thanks to the shared memory, granting access to the information across threads. Within this parallelized segment, each thread processed designated chunks of individuals, updating their positions based on the *population* and *weights* of nearby nodes. The *probability* vector for an individual deciding where to move is generated by *std::discrete_distribution(W)*, where *W* is the calculated weight vector based on section 2.2. This step is important for simulating the dynamic and random movement of individuals within the graph, considering the nearby population density and distance at each location.

The last parallel region simulates the infection dynamics within the population. Here is where we simulate the spread of disease from infected individuals to others. To simulate the correct probability of whether an uninfected individual will be infected, we first locate the infected ones and try to infect others who haven't been infected in the same nodes using a 13% infection rate based on section 2.1. Parallelization is achieved by distributing the workload among threads, specifically parallelizing the outer loop responsible for iterating through individuals. This strategy allows for an efficient exploration of the population, simulating infection attempts concurrently.

The results, including execution times and epidemic statistics for both the serial and parallel simulations, are printed out with statistics after each epoch. This allows for a direct comparison of the computational efficiency of the two implementations.

5. Computation results

The figure below shows a comparison between the simulation results using a serial approach and a parallel version with a variety of threads.



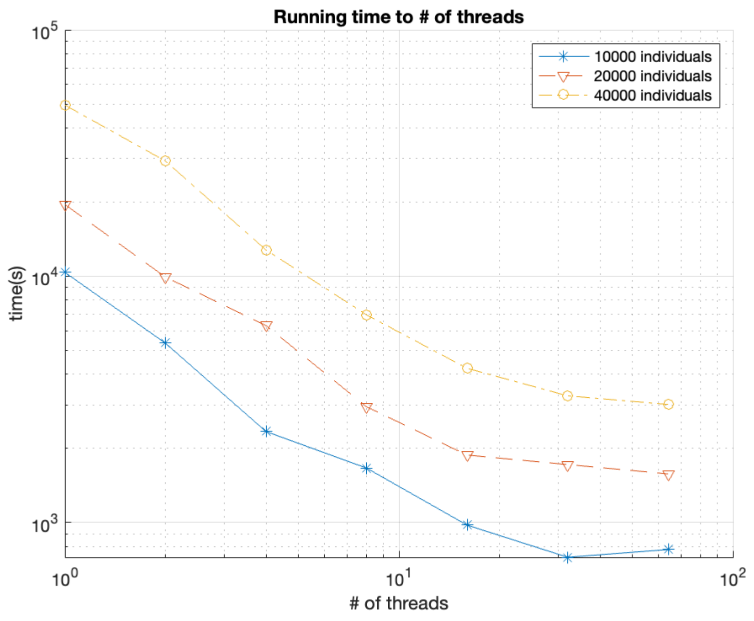
There are several random factors in our simulation. The simulation is initiated by randomly choosing 15 people to infect, the movement of these individuals is a random process, and the infection of others is also random. As a result, it is impossible to ask for the same result for each simulation. However, the results collectively still show a similar trend and exhibit a similar growth rate in the infected number. Still, undeniably, in a marginal case where infected individuals are initially set to be very far away from the healthy, the result is going to behave very differently since we require personnel to meet another infected one for the plague to spread in our code.

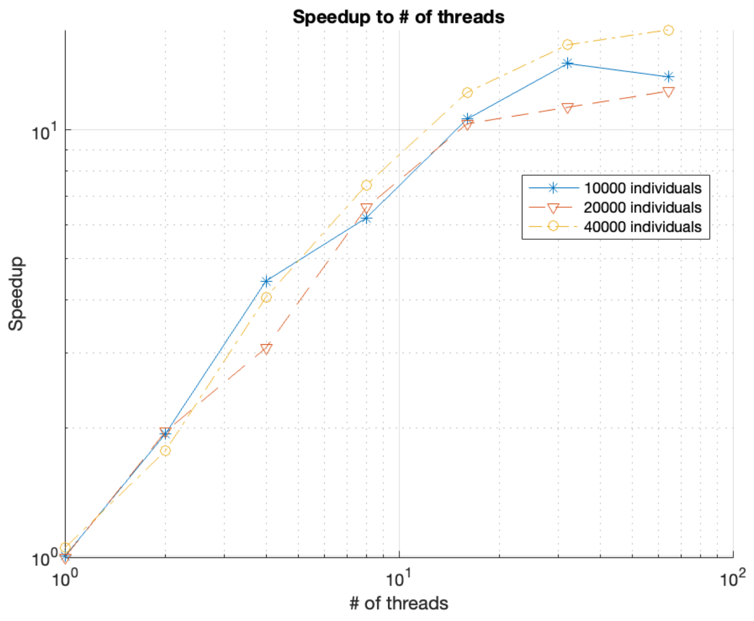
6. Timing analysis

6.1. Timing of different problem sizes

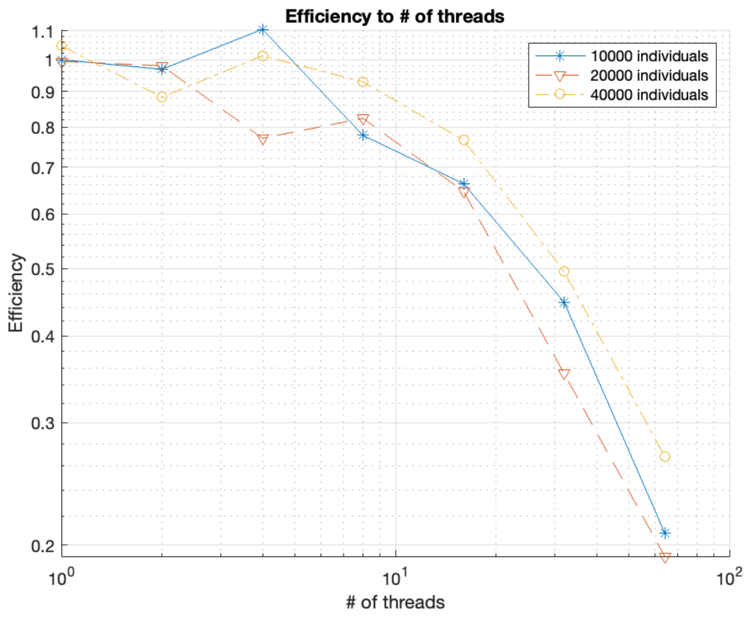
Problem Size \ # of threads	Nodes = 500 People = 10000	Nodes = 500 People = 20000	Nodes = 1000 People = 10000	Nodes = 1000 People = 20000	Nodes = 2000 People = 20000	Nodes = 2000 People = 40000
Serial	9407.25	19541.6	10366.6	19528.2	19556.3	49531.8
1	11139.8	25902.9	10364.7	21456	20447.8	51830.2
2	4738.78	9952.56	5350.02	12529.5	9920.27	29344.2
4	2481.52	5923.64	2342.97	5105.07	6309.33	12791.5
8	1375.22	2925.56	1665.16	3328.57	2952.96	6971.21
16	857.389	1909.2	977.208	1664.51	1880.34	4230.65
32	715.732	1600.5	723.776	1395.7	1720.28	3270.51
64	916.855	1695.76	777.707	1428.23	1576.93	3016.99

From our experiment results, the size of nodes has little impact on the running time. This is due to the parallelization we've designed focusing on dividing people into small chunks and allocating them to multiple threads for subprocessing. Thus, the timing of parallelization depends majorly on the number of people and threads. In general, the time of the serial version is slightly shorter than the parallel version with 1 thread. As more threads are added in, the overall time decreases roughly linearly according to the number of threads except for 64 threads, where Amdahl's law kicks in.





The speedup of our parallel code started to be confined by the serial part near 32 threads. This is predicted by Amdahl's law and is observed in Figure 2 and Figure 4, where the decrease in running time and the increase in speedup started to slow down and deviate from linear scaling.



Note that despite our treatment to eliminate the effect of random factors, it is still possible for the efficiency to exceed unity.

6.2 Timing by stage

Stage # \ # of threads	Stage 1 (statistics)	Stage 2 (population)	Stage 3 (move)	Stage 4 (infect)
Serial	0.930902	8.40616	11113.5	8321.06
1	1.2719	8.55198	11211.7	8215.25
2	1.1436	5.8058	5608.54	4113.54
4	0.94129	3.5504	2806.67	3133.53
8	0.72451	2.0728	1412.7	1048.73

Stage # \ # of threads	Stage 1 (statistics)	Stage 2 (population)	Stage 3 (move)	Stage 4 (infect)
Serial	0.930902	8.40616	11113.5	8321.06
1	1.2719	8.55198	11211.7	8215.25
2	1.1436	5.8058	5608.54	4113.54
16	0.846547	1.4887	723.867	522.914
32	1.66083	1.01882	411.086	332.891
64	6.27544	1.63349	547.977	355.996

To further analyze time consumption of each stage, we take the case with 2000 nodes and 20000 people as the example above.

From the results of Stage 1, it is observed that the computation in this stage does not contribute much to the total elapsed time. The timing from the serial version to 16 threads has no significant difference. However, in 32 and 64 threads, the timing has noticeably increased. We think the computational workload is too small to justify dividing it into 32 and 64 sub-jobs, as the dominant factor in timing appears to be the overhead associated with initializing sub-threads.

Stage 3 is where each individual decides on movement based on the population and weights of current and nearby nodes. This part of computation dominates the whole process, and this is where we can see the power of parallel computing.

In Stages 2 through 4, the time taken drops down by a factor of around 1.5 to 2 each time the number of threads is doubled. The times did not seem to reach a ceiling because we did not include any serial code in between the timing instructions. Thus, Amdahl's law did not come into play when we analyzed the time consumed by the parallel regions only.

7. Conclusion & Feedback

Based on our analysis above, our project demonstrates not only the power but also the constraints of parallel computing. This work provides valuable insights into the potential benefits of parallelization for plague-spreading modeling simulations and might aid researchers in related domains. The proposed program structure is a simple showcase of how parallelization can simulate real-world scenarios. The current code is capable of simulating plagues in a city, and it is easy to fit any configuration of landscapes in this framework because of the usage of graphs. Further expansion of this simulation includes adding vehicles by shortening the ways between nodes.

We appreciate this opportunity to participate in Parallel Computing. One of the interesting things we discovered during our work happens in section 6.2. At first, we set –*ntasks-per-node*=32 and –*nodes*=2 in our sbatch file. Then, the timing of Stage 1 drastically jumps to 103.01 ms in 32 threads and 110 ms in 64 threads. We believe it is because of cross-node communication mentioned in one of the previous discussions on Piazza. According to the discussion, using full threads in a node will occur with overhead that adds extra time consumption. Thus, the results in the chart reflect the disappointing performance when the thread count is 32 and 64. To fix the issues, we later set *–ntasks-per-node* to its maximum of 36 and *–nodes*=2 in our sbatch file. From the results of 32 and 64 threads in stage 1, the time becomes more reasonable but still acquires additional overhead in 64 threads.