*Proceedings of the 2021 Winter Simulation Conference*
*S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.*

# CSonNet: AN AGENT-BASED MODELING SOFTWARE SYSTEM FOR DISCRETE TIME SIMULATION

Joshua D. Priest
Aparna Kishore
Lucas Machi
Chris J. Kuhlman
Dustin Machi
S. S. Ravi

Biocomplexity Institute & Initiative
University of Virginia
1827 University Avenue
Charlottesville, VA 22904, USA

## ABSTRACT

Contagion dynamics on networks are used to study many problems, including disease and virus epidemics, incarceration, obesity, protests and rebellions, needle sharing in drug use, and hurricane and other natural disaster events. Simulators to study these problems range from smaller-scale serial codes to large-scale distributed systems. In recent years, Python-based simulation systems have been built. In this work, we describe a new Python-based agent-based simulator called CSonNet. It differs from codes such as Epidemics on Networks in that it performs discrete time simulations based on the graph dynamical systems formalism. CSonNet is a parallel code; it implements concurrency through an embarrassingly parallel approach of running multiple simulation instances on a user-specified number of forked processes. It has a modeling framework whereby agent models are composed using a set of pre-defined state transition rules. We provide strong-scaling performance results and case studies to illustrate its features.

## 1 INTRODUCTION

### 1.1 Background and Motivation

Computing contagion dynamics on networks, through simulation, quantifies interactions among members or agents of a population. In these models, nodes or vertices of networks often represent individuals, and edges represent pairwise interactions among vertices. However, the strength of network-based approaches is that the network model is abstract, and hence vertices can represent any entity, such as families (Yang et al. 2019), with appropriate semantics for edges. These simulation models have been applied to problems involving worm transmission in computer networks (del Rey 2013), social media and influence (Romero et al. 2011), and evacuation planning (Zia et al. 2012).

Several software systems exist for computing these contagion dynamics on networks, ranging from smaller-scale systems (Tisue and Wilensky 2004) to large-scale systems (Collier and North 2012), with respect to the sizes of problems that such systems can handle. For purposes of discussion, we define smaller networks as graphs with tens of nodes to 1000 nodes; intermediate networks as graphs with $10^3$ up to $10^6$ nodes; and large networks with numbers of nodes greater than $10^6$ (consistent with (Zia et al. 2012)). The importance of smaller and intermediate-sized simulation systems is attested to by the popularity of systems such as NetLogo (Tisue and Wilensky 2004) and Epidemics on Networks (EoN) (Miller and Ting 2019). These and larger-scale agent-based simulators are discussed in Section 2 (Related Work).

In this work, we describe CSonNet (Contagion Simulation ON NETworks), a new all-Python agent-based simulation (ABS) software for computing contagion dynamics on networked populations. CSonNet is a compromise between small-scale network-based contagion simulation systems and large-scale systems. The novelty of this work is described in the next subsection, which is followed by our contributions.

## 1.2 Novelty of Our Work

Our work presents a new simulation system for discrete time, network-based contagion dynamics that is suitable for graphs with about $10^5$ nodes. First, we introduce a new approach to building models of agent behaviors that uses an extensible set of state transition mechanisms. These mechanisms can be composed to yield several well-known models, and new ones. Second, we parallelize the simulator by forking worker processes that compute simulation instances, to achieve speedups that enable the system to handle intermediate to large networks with sufficiently fast execution times. Specifically, our goal is to drive down the total simulation time to about 100 seconds or less, for, say, roughly 100 replicates, so that the code can be used to run collections of simulations that involve hundreds of parameter sets. Third, CSonNet will be part of a first-of-its-kind cyberinfrastructure (CI) for network science, called *net.science* (Ahmed et al. 2020). The initial version of *net.science* was released in June 2021 (URL: https://net.science) and will be updated at regular intervals. This CI will have many features, including the ability to run agent-based simulations through CSonNet.

## 1.3 Our Contributions

**1. CSonNet Python agent-based simulator.** The simulation system performs discrete time simulations on populations that are represented as networks, where nodes represent entities (such as persons and organizations) and edges represent pairwise interactions between entities. The simulation module permits both serial and parallel execution modes. Parallelism is implemented using an embarrassingly parallel approach. Specifically, one simulation is typically composed of roughly 50 to 100 runs (or iterations), where each run is a separate instance. The instances are divided among, and run on, worker processes. In this work, we specify one worker process per core of a multicore commodity hardware node; however, one can assign multiple worker processes per core. The main process waits for the completion of the worker processes (which are forked processes in Python), and writes the results to files. CSonNet is designed to be light-weight so that it can be run on laptops and high performance computing (HPC) clusters, with minimal setup. Here, our focus is on evaluating the system on networks with up to $10^5$ nodes; in an expanded version, we will study scalability to larger networks. The current version of CSonNet supports undirected networks; extensions to directed networks are currently in progress. By late 2021/early 2022, we expect to make CSonNet public through a GitHub repository.

**2. Construction and extensibility of agent models.** A key feature of CSonNet, to make it versatile for a wide range of problems, is to form agent (behavior) models through mechanism composition. That is, an extensible set of state transition mechanisms is provided. These mechanisms are composed in user-specified ways to form models; users provide these specifications through configuration files. For example, in susceptible-infectious-recovered (SIR) models, an agent's state transition $S \rightarrow I$ is dictated by neighbor influence (i.e., an infectious neighbor can infect a healthy ego node). An agent transition $I \rightarrow R$ is described by a timed transition (i.e., an agent spends a prescribed amount of time in the infectious state). By composing these two mechanisms, a user obtains an SIR model. Other SIR models can be formed with different mechanisms. The importance of this approach is that CSonNet is going to be exposed in the *net.science* CI. Consequently, users need the ability to compose novel agent-models, through web app screens within the CI, and we achieve this through mechanism composition.

**3. Performance evaluation of the simulation module.** The simulation module is run in serial and parallel execution modes to evaluate the performance improvement due to concurrency. Strong-scaling studies are performed on four networks. These studies were performed on Dell PowerEdge C6420 2.666 GHz hardware nodes, with 384 GB RAM and 40 cores per node. Each core in a node is an Intel Xeon Gold 6148, 2.40 GHz processor with 1280 KiB L1 cache, 20 MiB L2 cache, and 27 MiB L3 cache. We find that strong scaling continues out through 32 worker processes and cores and that simulations of 100 runs (i.e., 100 diffusion instances) with 100 time steps per run, on a 75,877-node network (Epinions) can be completed in 62.5 seconds. The numbers of runs and time steps are fairly onerous; see Section 6. Amortized over the 100 iterations, this is 0.625 seconds per iteration (i.e., one simulation instance from one set of initial conditions). The execution time for 32 worker processes represents a factor 15.8 speedup over the serial code. In the ideal case, one would expect a speedup of about 32. The observed speedup factor of 15.8 is due to Python's implementation of multi-processing where a worker process is not implemented as a lightweight threading process which uses the same address space as the parent process. Instead, Python implements multi-processing concurrency through heavyweight forking of processes that requires a complete code and data image. The execution times observed in our performance study demonstrate that hundreds of simulations can easily be run in a day on one compute node, which is one of our goals.

In addition to the above, the paper also includes three cases studies. One uses a Granovetter (1978) threshold model and two use an SIR model. These case studies encompass many simulations to investigate the effects of model parameter values and initial conditions.

**Paper organization.** Section 2 contains related work. Section 3 presents the graph dynamical systems formalism which the simulation system implements. Section 4 describes the simulation system. Section 5 provides information about the networks used in our experiments. Section 6 provides performance evaluation in terms of strong scaling, and Section 7 contains case studies for problems that use the CSonNet system. A summary is in Section 8.

## 2 RELATED WORK

Many simulators for agent-based models (ABMs) have been developed. Abar et al. (2017) provide a detailed review of about eighty five simulators that are used in various contexts. Rossetti et al. (2018) provide an overview of some of these simulators and compare their performance. The simulators differ in many aspects, including development techniques, programming language, purpose, scalability, usability, and performance.

Many recent simulation systems have been developed using Python; these systems exploit the large library of graph algorithms available in NetworkX. For example, the NDLIB simulator (Rossetti et al. 2018) supports several standard epidemic models (such as SIR, SIS, etc.); it also includes a simulation server that allows remote execution of experiments. Miller and Teng (2019) discuss the Epidemics on Networks (EoN) system which was developed for simulating small-scale epidemic models; it supports several popular epidemic models, including SIS and SIR. Epigrass (Coelho et al. 2008) is mainly intended for large-scale simulations of epidemics using metapopulation models. Nepidemix[1] supports most common epidemic models and allows a user to save intermediate results (such as disease prevalence and state transitions) that can provide additional insights about the progression of an epidemic. Epydemics (Dobson 2020) is built on top of the epyc experiment management library which allows users to run simulations on individual machines, multicore systems and parallel computing clusters. Mesa[2] is another Python framework for agent-based modeling. New models are added to this framework through Python modules. In contrast, new models are specified in CSonNet through the composition of state transition rules.

Abar et al. (2017) also discuss a number of small- and medium-scale agent-based simulators (e.g., AgentScript, FLAME, NetLogo). Among these, Netlogo (Railsback, Ayllón, Berger, Grimm, Lytinen, Sheppard, and Thiele 2017) is a popular tool; however, it is not geared for simulations of large social networks using high-performance computing (HPC) systems. Repast (HPC version)[3], discussed by Collier

[1]http://nepidemix.irmacs.sfu.ca

[2]https://mesa.readthedocs.io/en/stable/

[3]http://repast.sourceforge.net/

and North (2012), is an agent-based modeling framework, where behavior models can be built using high-level programming abstractions. Swarm[4] is an advanced parallel discrete event simulation (PDES) framework, where behavior models are written in Objective-C and Java. See (Allan 2010) for additional details regarding these and other frameworks.

## 3 GRAPH DYNAMICAL SYSTEMS

The underlying model implemented in our simulation system CSonNet is called a **graph dynamical system** (GDS). A GDS is a powerful formal model that can simulate Turing machines for specific complexity classes (Barrett et al. 2011; Barrett et al. 2006; Adiga et al. 2019). We start with the formal model.

### 3.1 GDS Formalism

A GDS (Mortveit and Reidys 2007), denoted $\mathsf{S}(G, \mathsf{F}, \mathsf{K}, \mathsf{R})$, is comprised of four elements: $(i)$ a network $G(V, E)$, where $V$ and $E$ denote the vertex and edge sets; $(ii)$ a sequence $\mathsf{F}$ of vertex functions; $(iii)$ a set $\mathsf{K}$ of vertex states; and $(iv)$ a specification $\mathsf{R}$ of the order in which vertex functions are executed.

The graph $G(V, E)$ represents the interaction network. The vertex set $V$ and edge set $E$ have cardinalities $n = |V|$ and $m = |E|$. Agents are vertices and pairwise agent interactions are edges in $G$. In general, edges may be directed or undirected. In this paper, for simplicity, we will consider graphs with undirected edges.

Each agent $v_i \in V$, $i \in \{1, 2, \ldots, n-1, n\}$, has an **agent state** or **vertex state** $s_i \in \mathsf{K}$. For many models, such as independent cascade or linear threshold (Kempe et al. 2003), a Boolean state set is used; i.e., $\mathsf{K} = \mathbb{B} = \{0, 1\}$, where 0 (respectively, 1), is typically an inactive (respectively, active) state. One epidemic model is the susceptible, infectious, recovered (SIR) system in which $\mathsf{K} = \{S, I, R\}$. The **system state** (also called a **configuration**) $s = (s_1, s_2, \ldots, s_n)$ is the sequence of vertex states. Let the **local states**, denoted by $s[v_i]$, represent the sequence of states of vertex $v_i$ and all of its distance-1 neighbors in $G$. These quantities, at time $t$, are denoted by $s_i^t$, $s^t$, and $s^t[v_i]$ respectively.

Each agent $v_i$ has a **local function** $f_i \in \mathsf{F}$ that quantifies its state transitions. The state of agent $v_i$ at time $t+1$ is given by

$$s_i^{t+1} = f_i(s^t[v_i]) \ . \tag{1}$$

Thus, the next state of $v_i$ is a function of the current state of $v_i$ and those of its distance-1 neighbors in $G$.

Hence, given an **initial system state** (**initial configuration**) at time $t = 0$, the $f_i$ for all $i \in \{1, 2, \ldots, n\}$ are evaluated simultaneously at each $t \in (1, 2, \ldots, t_{max})$, up through some specified time $t_{max}$. This is a **simulation instance**.

In this work, we assume that specification of update order $\mathsf{R}$ corresponds to the **synchronous** update scheme; that is, all agents evaluate their local functions $f_i$ and update their states *simultaneously* at each time step; see Equation (1). Among other things, this enables greater parallelization of simulation computations, leading to more efficient calculations. We now provide an example to make these ideas concrete.

### 3.2 Example GDS for Epidemic Simulation

Consider the 7-vertex binary tree of Figure 1, wherein all edges are undirected. Each node is a person, and edges represent face-to-face interactions between pairs of nodes. We model an SIR epidemic process on this tree. Each vertex is in one of the states $\mathsf{K} = \{S, I, R\}$, where $S$ is susceptible, $I$ is infectious, and $R$ is recovered. Each infected person is infectious for $t_{inf} = 2$ days, after which she transitions to the recovered state. We assume a uniform edge weight (probability) $w_e = 0.05$ for all edges $\{v_j, v_k\}$ such that if $v_j$ is in state $s_j = I$ and $v_k$ is in state $s_k = S$, then the probability that $v_k$ contracts the virus and becomes infectious in the next time step is $Pr[s_k^{t+1} = I | s_j^t = I, s_k^t = S] = w_e$.

To evaluate whether a node $v_k$ in state $S$ transitions under these circumstances, we use the following scheme. A Bernoulli trial is performed for each neighbor $v_j$ of $v_k$ where $s_j = I$. Assume that $v_k$ has $q$

[4]http://savannah.nongnu.org/projects/swarm

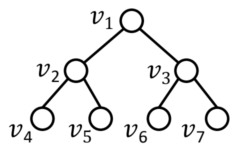


Figure 1: Network $G(V,E)$ with $n = |V| = 7$ and $m = |E| = 6$ undirected edges.

neighbors. That is, at each time $t$, a number $0 < p \leq 1$ is drawn uniformly randomly for each neighbor $v_j$ with $s_j = I$, and if $p \leq w_e$, this is a *positive* Bernoulli trial (PBT); otherwise, it is a *negative* Bernoulli trial (NBT). If any of the trials is a PBT, then $v_k$ transitions state from $S$ to $I$, i.e., $S \rightarrow I$. Otherwise, (i.e., all of the up-to $q$ trials are NBTs), $v_k$ remains in state $S$. Since the SIR model is stochastic, we describe the dynamics in our example for a specific sequence of PBTs and NBTs.

Figure 2 provides the dynamics over time steps $t = 0$ through $t_{max} = 5$. The colors of the states $S$, $I$, and $R$ are white, black, and gray, respectively. The initial conditions—the states of vertices at $t = 0$—are that all agents are in state $S$, except $v_2$, which is in state $I$. At $t = 1$, Bernoulli trials are performed for $v_1$, $v_4$, and $v_5$ and they are all NBTs, so no state change occurs. For $t = 2$, there are PBTs for $v_1$ and $v_5$, but there is an NBT for $v_4$. Thus, $v_1$ and $v_5$ transition $S \rightarrow I$ at the end of $t = 2$, and $v_4$ remains in state $S$. Also, at the end of $t = 2$, $v_2$ transitions $I \rightarrow R$ because $t_{inf} = 2$. Note the subtlety: $v_2$ is infectious *during* $t = 2$, but at the *end* of this time period (which is synonymous with the starting conditions for $t = 3$), $v_2$ is recovered—it is the end of the second time period and the end of the infectious period.

Vertices $v_1$ and $v_5$ will be infectious at $t = 3$ and at $t = 4$, but at the end of $t = 4$, they transition to $R$. During $t = 3$, $v_1$ infects $v_3$ owing to a PBT, and $v_3$ becomes infectious at the end of $t = 3$. During $t = 4$ and $t = 5$, $v_6$ and $v_7$ have only NBTs with respect to the infectious agent $v_3$ and hence these former two agents do not transition; they remain in state $S$. Because all vertices are either in state $S$ or $R$, no more transitions are possible, and this state is a *fixed point* for these dynamics. It is important to note that the contagion dynamics of a GDS are the results of computations of a simulation.

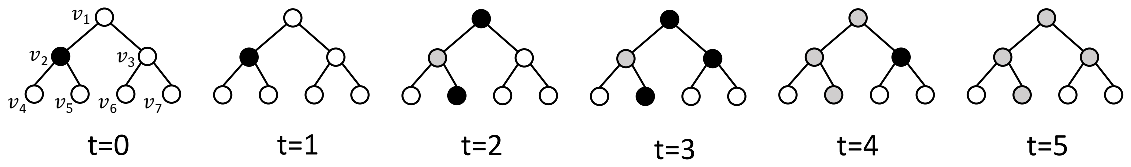


Figure 2: Contagion dynamics using an SIR model at each $v_i \in V$, where the weight of each edge is $w_e$. The states of nodes $S$, $I$, and $R$ are colored white, black, and gray, respectively. At time $t = 0$, there is one seed node $v_2$ in the infected state, i.e., $s_2 = I$, and all other nodes are susceptible, i.e., for $v_i$ with $i \in \{1,3,4,5,6,7\}$, $s_i = S$. The infectious duration is $t_{inf} = 2$ days.

## 4 SYSTEM DESCRIPTION

### 4.1 Overview and Algorithms

CSonNet is a discrete time, agent-based simulator (ABS) that computes contagion dynamics on networked populations. CSonNet implements the GDS framework of Section 3. A **simulation** consists of a user-specified number of iterations, and the number of time steps for each iteration. An **iteration** consists of assigning initial states $s_i^t$ $(t = 0)$ to all nodes $v_i \in V$ in the network $G$, assigning a state transition model to each $v_i$, and running time forward in discrete steps until a maximum number $t_{max}$ of steps is completed. The state transition model $M_i^{st}$ for $v_i$ is the local function $f_i$ discussed in Section 3.1. The example in Section 3.2 is an iteration that is executed for $t_{max} = 5$ time steps.

**Algorithm 1:** Simulation Process (Main Process)

```
1 Input: Number of iterations $n_i$. Maximum number of time steps $t_{max}$ per iteration. Name of the file
    containing the network $G(V,E)$. Name of the file containing the initial conditions $\mathbb{I}$ for every node in each
    iteration. Name of the file containing the state transition model $M_i^{st}$ for each $v_i \in V$. Number $n_{wp}$ of worker
    processes to fork.
2 Output: For each iteration and each time step $t$ of each iteration, the list of nodes $v_i$ that transition states
    and their new states $s_i^t$.
3 Steps:
    A.  Read the file containing the network $G(V,E)$.
    B.  Read the file containing the initial conditions $\mathbb{I}$.
    C.  Read the file containing the state transition models $M^{st}$.
    D.  Create a pool of $n_{wp}$ worker processes.
    E.  For each iteration $\eta \in [1,\ n_i]$:
          1. Assign $M^{st}$, $G$, the initial conditions from $\mathbb{I}$ for this iteration, $\eta$, and $t_{max}$ to an available worker process,
             and start the worker process execution on this iteration, i.e., invoke Algorithm 2.
          2. Receive timing data and output file name from the worker process upon its completion of this iteration.
    F.  Concatenate the output files generated by each of worker processes over all iterations.
    G.  Print timing data.
```

Algorithms for the main process and a worker process are given in Algorithm 1 and Algorithm 2, respectively. Concurrency in CSonNet is implemented by forking worker processes. The CSonNet main process reads the files containing the network $G$, initial conditions $\mathbb{I}$ file, and the state transition models $M^{st}$. It then forks a user-specified number of worker processes. Each worker process is allocated iterations by the main process, one at a time; when an iteration completes, the main process assigns another iteration to that process. This results in slightly better load balancing than assigning up-front an appropriate number of iterations to each worker process. For a given number $n_{wp}$ of worker processes and a specified number $n_i$ of iterations, the expected number of iterations per worker process is at most $\lceil n_i/n_{wp} \rceil$.

**Algorithm 2:** Worker Process

```
1 Input: Iteration number $\eta$. Maximum number of time steps $t_{max}$. Network $G(V,E)$. Initial condition (i.e.,
    initial state) for every node in this iteration. State transition models $M^{st}$.
2 Output: $(i)$ For this iteration and for each time step $t$, the list of nodes $v_i$ that transitions states and their new
    states $s_i^t$. These data values are written to a file, named by iteration number $\eta$. $(ii)$ Timing data for this
    iteration.
3 Steps:
    A.  Start timer for this iteration.
    B.  Open the new output file for this iteration.
    C.  Assign initial state $s_i^t$ at $t = 0$ to each $v_i \in V$, from $\mathbb{I}$. Write initial states to output file.
    D.  For each time $t \in [0,\ t_{max} - 1]$:
          1. For each node $v_i \in V$:
               a. Compute the next state $s_i^{t+1}$ of $v_i$, at time $t+1$, according to Equation (1).
               b. If $v_i$ has changed state (i.e., $s_i^t \neq s_i^{t+1}$), then write $\eta$, $t+1$, $v_i$, $s_i^{t+1}$ to file.
    E.  Stop timer.
    F.  Return timing data and output file name to main process.
```

Table 1: Illustrations of selected well-known state transition models $M^{st}$ in CSonNet. For each model, there is one or more state transition rules.

| Name of Model | State Transition | State Transition Rule |
|---|---|---|
| SIR | S → I | Probability influence |
| | I → R | Probability |
| SIR | S → I | Probability influence |
| | I → R | Discrete |
| Deterministic Threshold | 0 → 1 | Threshold |

### 4.2 Agent State Transition Models

Table 1 lists some state transition models in CSonNet. In some cases, a given model can be implemented in different ways. The SIR model, as an example, can be implemented where an agent transitions from state I to state R either stochastically (i.e., on each time step, there is a probability of this transition), which corresponds to state transition rule of *Probability* in the table, or after a specific number of time steps, which corresponds to state transition rule *Discrete* in the table. These and other models are specified within input files to the simulation code; no code changes are made. The specification of state transition models through configuration files is an important feature of the system. We present these models because we use them in the case studies of Section 7.

## 5 NETWORKS

We used four networks ranging in size from about 200 nodes to 76,000 nodes in our performance evaluation and case studies. The list of these networks along with their structural properties is shown in Table 2. These properties were generated with the codes in the *net.science* cyberinfrastructure (Ahmed et al. 2020). which contains SNAP (Leskovec and Sosič 2016) and NetworkX (Hagberg et al. 2008) codes.

Table 2: Networks used in performance evaluation and case studies. If there are multiple connected components in a graph, we use only the giant component. Here, $n$ and $m$ are numbers of vertices and edges, respectively, in the giant component; $d_{ave}$ and $d_{max}$ are average and maximum degrees; $k_{max}$ is the maximum k-core; $c_{ave}$ is average clustering coefficient; and $\Delta$ is graph diameter.

| Network | Type/Domain | Edge Direction | $n$ | $m$ | $d_{ave}$ | $d_{max}$ | $k_{max}$ | $c_{ave}$ | $\Delta$ |
|---|---|---|---|---|---|---|---|---|---|
| Jazz | interaction | undirected | 198 | 2742 | 27.7 | 100 | 29 | 0.617 | 6 |
| Ca-Hepth | collaboration | undirected | 8638 | 24806 | 5.74 | 65 | 31 | 0.482 | 18 |
| Ca-Astroph | collaboration | undirected | 17903 | 196972 | 22.0 | 504 | 56 | 0.633 | 14 |
| Epinions | trust on epinions | undirected | 75877 | 405739 | 10.7 | 3044 | 67 | 0.138 | 14 |

## 6 STRONG SCALING PERFORMANCE

We address strong scaling, i.e., how execution time decreases with increasing numbers of compute cores, for a fixed problem size. Strong scaling results from running CSonNet on the four networks of Table 2 are provided in Figure 3. In both plots, the times are the durations in seconds to complete 100 iterations of 100 time steps each for an SIR model. Numbers of seed nodes used in each iteration for each network are given in the figure caption; the values were chosen to generate wide-spread contagion diffusion. For each network, simulations are performed with numbers $n_{wp}$ of worker processes varying from one to 32 (our hardware nodes have 40 cores each). In Figure 3, the total end-to-end execution time is plotted on the y-axis. It is seen that for a given network, the execution time decreases as the number of worker processes increases. Also, execution times increase with network size. The data show that 100 iterations of 100 time steps each can be run on the 75,877-node Epinions network in 62.5 seconds.

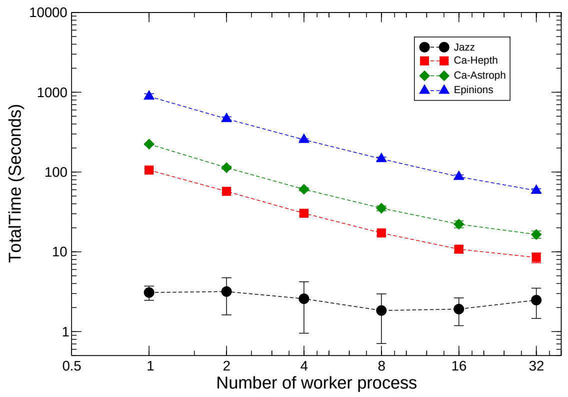


Figure 3: Strong scaling results for CSonNet, where execution time (y-axis) is the total end-to-end execution time. Data are generated on four networks. Each simulation is run for 100 iterations, with each iteration consisting of 100 time steps, using an SIR model with $t_{inf} = 12$ and $w_e = 0.03$. Number of seed nodes $n_s = 10$ for Jazz, 400 for Ca-Hepth, 500 for Ca-Astroph, and 2000 for Epinions. Each data point is the average of ten measured values. Error bars at $\pm$ one standard deviation indicate that the measured times for a set of conditions are within a tight range. In these simulations, one worker process is assigned to a dedicated compute core of one compute node. Results show that CSonNet exhibits strong scaling. For the behavior of the Jazz network, see text.

In Figure 3, the data for Jazz does not demonstrate strong scaling as clearly as the other networks. This is because the network is small; therefore, the execution times are on the same order as delays in starting worker processes, giving rise to noisy data.

One hundred iterations was chosen because there is variability in initial conditions or model parameters that are investigated through many iterations. One hundred time steps is somewhat arbitrary, but for epidemic simulations, for example, a 3-month (e.g., roughly 100-day) prediction window wherein each time step is one day, is generally considered a long-running simulation. (Typically, one recalibrates parameters and runs for much shorter windows, on the order of a week or two.) Thus, the execution times provided here are for fairly conservative conditions. Table 3 provides representative breakdowns of execution times into component parts.

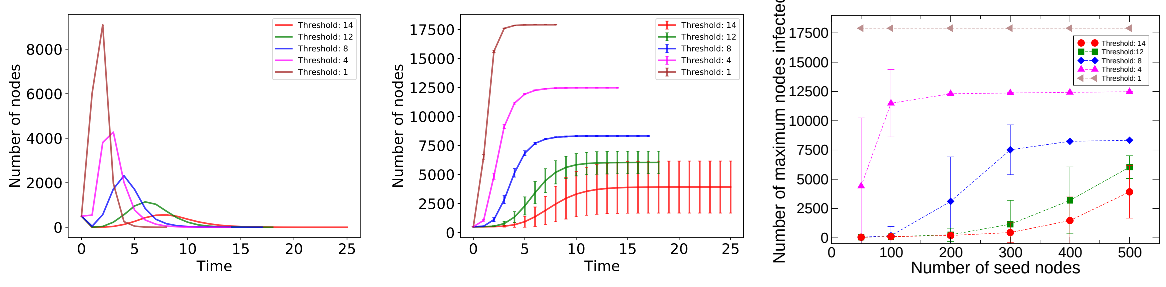
Table 3: Selected timing study data to show relative times for computation versus overhead or setup operations. The setup times will be independent of number $n_{wp}$ of worker processes. The total setup time (last column), including reading in the graph (second-to-last column) are both small in comparison to the worker process execution times.

| **Network** | **No. Worker Processes** | **Total Execution Time(s)** | **Total Worker Process Time(s)** | **Time to Read Graph(s)** | **Total Setup Time(s)** |
|---|---|---|---|---|---|
| Jazz | 16 | 2.86 | 1.72 | 0.0064 | 0.040 |
| Ca-Hepth | 16 | 12.17 | 10.46 | 0.069 | 0.702 |
| Ca-Astroph | 16 | 25.25 | 22.18 | 0.49 | 1.82 |
| Epinions | 16 | 93.77 | 82.77 | 1.27 | 7.38 |

## 7 CASE STUDIES

### 7.1 Case Study 1: Threshold Model on the Ca-Astroph Network

This is a study of the Granovetter (1978) threshold model to simulate the spreading of information or opinions on the Ca-Astroph network to produce collective action, for several threshold values between $\theta = 1$ to 14. Five simulations of 100 iterations each are completed with one simulation for each threshold value. (All nodes have the same threshold for all iterations of one simulation.) Nodes are in state 0 (inactive) or state 1 (active). In each of the 100 iterations, $n_s = 500$ nodes are uniformly at random set to state 1 at $t = 0$, with the remaining nodes in state 0. There is a different seed set for each iteration. A node can transition state $0 \rightarrow 1$, but once in state 1, it remains active. A node transitions to state 1 when at least a threshold $\theta$ number of its neighbors are already in state 1. Figure 4a shows the number of newly active (i.e., state 1) nodes as a function of time; such plots are commonly referred to as epi-curves. As the threshold increases, the peak in numbers of newly active nodes decreases and is pushed out to greater times. The cumulative number of nodes in state 1 is provided in Figure 4b. Error bars provide $\pm$ one standard deviation. For lesser thresholds, contagion spreads so quickly that the variability in results across the 100 iterations is very small. The variation increases as $\theta$ increases. Since the threshold model is deterministic, variability comes from the different seed sets over the 100 iterations. In Figure 4c, the final number of active nodes is plotted against the number $n_s$ of seed nodes; in this plot, $n_s$ varies from 50 to 500. The results show a pronounced effect of $n_s$ for $\theta \geq 4$.

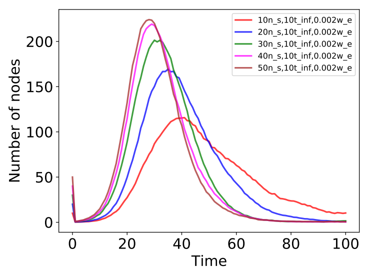


(a) Epi-curves (b) Cumulative infection curves (c) Final number of infected nodes

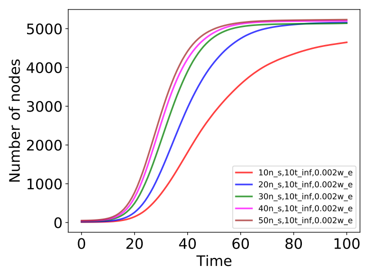
Figure 4: Simulation results for modeling information transmission on the Ca-Astroph network using the threshold model of (Granovetter 1978). There are $n_s = 500$ seed nodes per iteration in (a) and (b). Seed nodes are chosen uniformly at random without replacement from all nodes. (a) Epi-curves (i.e., the y-axis is the number of newly activated nodes at each time $t$) and (b) Cumulative infection curves (i.e., the y-axis is the cumulative number of activated nodes at each time $t$) for different threshold values from $\theta = 1$ to 14. (c) Final numbers of activated nodes as a function of the seed set size (50 to 500 seed nodes) and $\theta$. Error bars in (b) and (c) denote $\pm$ one standard deviation.

### 7.2 Case Study 2: SIR Model on the Epinions Network

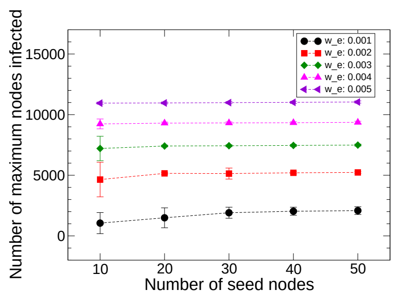
We study the diffusion of information on the Epinions network using an SIR model. This model simulates the behavior that people will only spread information for a finite time (the infectious duration $t_{inf} = 10$ days). During this "infectious" period, agents try to spread information to their susceptible neighbors across edges with probability $w_e = 0.002$. In Figures 5a and 5b, epi-curves and cumulative infection curves are provided for $w_e = 0.002$ and $n_s = 10$ to 50 nodes (10n to 50n in the legend). In Figure 5a, the number of newly infected nodes at $t = 0$ is the number of seed nodes, and then these numbers drop because far fewer nodes get infected at $t = 1$. Overall, spreading increases as $n_s$ increases. Figure 5c shows the final number of nodes that possess the information as edge weight increases from 0.001 to 0.005, for the same numbers of seed nodes. The results are relatively independent of numbers of seed nodes. But the effect of increasing edge weight $w_e$ is pronounced. Surprisingly, in Figure 5c at $n_s = 50$, the effect of $w_e$ on the total number of nodes that reaches the infected state $I$ appears to be close to linear, at least for $0.002 \leq w_e \leq 0.005$.



(a) Epi-curves



(b) Cumulative infection curves



(c) Final number of infected nodes

Figure 5: Simulation results for modeling information transmission on the Epinions network using an SIR model. Different numbers of seeds nodes are $n_s = 10$, 20, 30, 40, and 50. Seed nodes are chosen uniformly at random without replacement from all nodes. (a) Epi-curves (i.e., the y-axis is the number of newly infected nodes at each time $t$) and (b) Cumulative infection curves (i.e., the y-axis is the cumulative number of infected nodes at each time $t$) for different sizes of seed node sets. The infectious period is $t_{inf} = 10$ and the edge weight $w_e = 0.002$. (c) Final numbers of infected nodes as a function of seed set sizes and $w_e$ in the SIR model. The infectious period is $t_{inf} = 10$. Error bars denote $\pm$ one standard deviation.

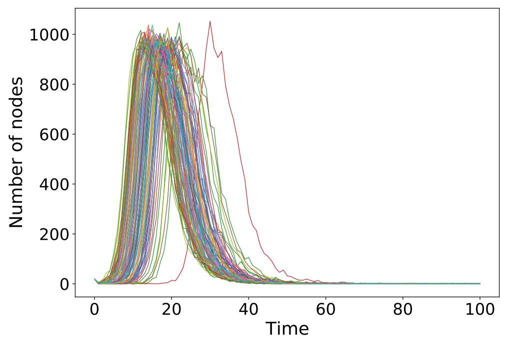
## 7.3 Case Study 3: Effect of Averaging SIR Model Results on the Epinions Network

The setup for this case study is very similar to the previous one: we use the same SIR model on the same network. Here, we use $w_e = 0.006$, which is greater than previous values. The point of this case study is to illustrate the effects of averaging the results of multiple iterations, which is routinely done. In Figure 6a, the 100 epi-curves are displayed, one for each iteration of a simulation. In Figure 6b, the time vs point-wise average curve resulting from the 100 curves is provided. Note that the average curve has a peak that is much less (roughly 80% of the maximum from individual iterations).

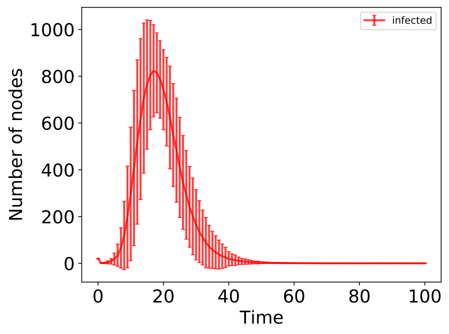
The importance of this result can be explained with an example. While we use an SIR model here in a social context because our graph Epinions is an online communication network, it has a power-law degree distribution like many social networks. If these results are applied to a disease epidemic, and the average epidemic curve in Figure 6b is used to predict the peak value of the number of hospital beds needed or of the number of ventilators needed for an epidemic like COVID-19, then the most extreme predicted outcome, as represented by the top of the red error bars at $+1$ standard deviation, only represents the *typical* outbreak generated by each individual epi-curve in Figure 6a. That is, there is no factor of safety in the results when considering the worst-case conditions as represented by $+1$ standard deviation.

## 8 SUMMARY AND FUTURE WORK

We presented the CSonNet discrete time simulator. Contributions of this work are summarized in Section 1.3. The system combines the ease of programming system extensions using Python with concurrency. The CSonNet system supports many state transition models; additional models will be presented in an expanded version of this work. Performance comparisons with other simulation systems (e.g., NDLIB (Rossetti, Milli, Rinzivillo, Sirbu, Pedreschi, and Giannotti 2018)) will also be presented in an expanded version of this paper. In this work, we focus on graph sizes where simulations can be completed in 100 seconds or



(a) 100 Epi-curves



(b) Final number of infected nodes

Figure 6: Simulation results for modeling information transmission on the Epinions network using an SIR model. The y-axes are the numbers of newly infected nodes at each time $t$. All 100 iterations are performed with one seed set where $n_s = 20$. In the SIR model, $t_{inf} = 10$ and $w_e = 0.006$. (a) 100 epi-curves (i.e., one epi-curve for each iteration) and (b) the average epi-curve over the 100 iterations, with $\pm$ one standard deviation error bars.

less, to accommodate large parametric studies that often involve roughly four dimensions, with, say, five values per dimension, resulting in $5^4 = 625$ simulations. We will investigate scaling to larger networks as part of future work.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. We thank our colleagues at NSSAC and Research Computing at The University of Virginia for providing computational resources and technical support. This work has been partially supported by University of Virginia Strategic Investment Fund award number SIF160, NSF Grant OAC-1916805 (CINES), NSF Grant CMMI-1916670 (CRISP 2.0) and CCF-1918656 (Expeditions).

## REFERENCES

Abar, S., G. K. Theodoropoulos, P. Lemarinier, and G. M. O'Hare. 2017. "Agent Based Modelling and Simulation Tools: A Review of the State-of-Art Software". *Computer Science Review* 24:13–33.

Adiga, A., C. J. Kuhlman, M. V. Marathe, H. S. Mortveit, S. S. Ravi, and A. Vullikanti. 2019. "Graphical Dynamical Systems and Their Applications to Bio-Social Systems". *Int J Adv Eng Sci Appl Math* 11:153–171.

Ahmed, N. K., R. A. Alo, C. T. Amelink, Y. Y. Baek, A. Chaudhary, K. Collins, A. C. Esterline, E. A. Fox, G. C. Fox, A. Hagberg, R. Kenyon, C. J. Kuhlman, J. Leskovec, D. Machi, M. V. Marathe, N. Meghanathan, Y. Miyazaki, J. Qiu, N. Ramakrishnan, S. S. Ravi, R. A. Rossi, R. Sosic, and G. von Laszewski. 2020. "net.science: A Cyber-infrastructure for Sustained Innovation in Network Science and Engineering". In *Gateway Conference*, 71–74.

Allan, R. J. 2010. *Survey of Agent Based Modelling and Simulation Tools*. Science & Technology Facilities Council New York.

Barrett, C. L., H. B. Hunt III, M. V. Marathe, S. Ravi, D. J. Rosenkrantz, and R. E. Stearns. 2006. "Complexity of Reachability Problems for Finite Discrete Dynamical Systems". *Journal of Computer and System Sciences* 72(8):1317–1345.

Barrett, C. L., H. B. H. III, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and R. E. Stearns. 2011. "Modeling and Analyzing Social Network Dynamics Using Stochastic Discrete Graphical Dynamical Systems". *Theoretical Computer Science* 412(30):3932–3946.

Coelho, F. C., O. G. Cruz, and C. T. Codeço. 2008. "Epigrass: a Tool to Study Disease Spread in Complex Networks". *Source code for biology and medicine* 3(1):1–9.

Collier, N., and M. North. 2012. “Parallel Agent-Based Simulation With Repast for High Performance Computing”. *Simulation* 89(10):1215–1235.

del Rey, Á. M. 2013. “A SIR e-Epidemic Model for Computer Worms Based on Cellular Automata”. In *Advances in Artificial Intelligence*, 228–238.

Dobson, S. 2020. “epydemic Documentation”.

Granovetter, M. 1978. “Threshold Models of Collective Behavior”. *The American Journal of Sociology* 83(6):1420–1443.

Hagberg, A. A., D. A. Schult, and P. J. Swart. 2008. “Exploring Network Structure, Dynamics, and Function Using NetworkX”. In *7th Python in Science Conference (SciPy2008)*, 11–15.

Kempe, D., J. Kleinberg, and E. Tardos. 2003. “Maximizing the Spread of Influence Through a Social Network”. In *Proc. ACM KDD*, 137–146.

Leskovec, J., and R. Sosič. 2016. “SNAP: A General-Purpose Network Analysis and Graph-Mining Library”. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8(1):1.

Miller, J. C., and T. Ting. 2019. “EoN (Epidemics on Networks): a Fast, Flexible Python Package for Simulation, Analytic Approximation, and Analysis of Epidemics on Networks”. *Journal of Open Source Software* 4(44):5.

Mortveit, H. S., and C. M. Reidys. 2007. *An Introduction to Sequential Dynamical Systems*. Universitext. Springer Verlag.

Railsback, S., D. Ayllón, U. Berger, V. Grimm, S. Lytinen, C. Sheppard, and J. Thiele. 2017. “Improving Execution Speed of Models Implemented in NetLogo”. *Journal of Artificial Societies and Social Simulation* 20(1).

Romero, D., B. Meeder, and J. Kleinberg. 2011. “Differences in the Mechanics of Information Diffusion”. In *WWW*, 695–704.

Rossetti, G., L. Milli, S. Rinzivillo, A. Sirbu, D. Pedreschi, and F. Giannotti. 2018. “NDlib: A Python Library to Model and Analyze Diffusion Processes Over Complex Networks”. *International Journal of Data Science and Analytics* 5:61–79.

Tisue, S., and U. Wilensky. 2004. “NetLogo: Design and Implementation of a Multi-Agent Modeling Environment”. In *SwarmFest Conference*, 1–17. Ann Arbor, May 2004.

Yang, Y., L. Mao, and S. S. Metcalf. 2019. “Diffusion of Hurricane Evacuation Behavior Through a Home-Workplace Social Network: A Spatially Explicit Agent-Based Simulation Model”. *Computers, Environment and Urban Systems* 74:13–22.

Zia, K., A. Riener, K. Farrahi, and A. Ferscha. 2012. “A New Opportunity to Urban Evacuation Analysis: Very Large Scale Simulations of Social Agent Systems in Repast HPC”. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, 233–242.

## AUTHOR BIOGRAPHIES

**JOSHUA D. PRIEST** has been recently graduated from the University of Virginia (UVA) with a Bachelor of Science Degree in Computer Science. His email address is jdp8jb@virginia.edu.

**APARNA KISHORE** is a PhD student in Computer Science at the UVA. Her email address is ak8mj@virginia.edu.

**LUCAS MACHI** is a student at New River Community College and is a member of the Biocomplexity Institute and Initiative (BII) of the UVA. His email address is lhm4v@virginia.edu.

**DUSTIN MACHI** is a Senior Software Architect at BII of UVA. His email address is dm8qs@virginia.edu.

**CHRIS J. KUHLMAN, S. S. RAVI** are faculty at BII of UVA. Their email addresses are [cjk8gx], [ssr6nh]@virginia.edu.