PanSim + Sim-2APL: A Framework for Large-Scale Distributed Simulation with Complex Agents

Parantapa Bhattacharya[1], A. Jan de Mooij[2], Davide Dell'Anna[2], Mehdi Dastani[2], Brian Logan[2], and Samarth Swarup[1]

[1] University of Virginia, Charlottesville VA 22904, USA
{parantapa,swarup}@virginia.edu
[2] Universiteit Utrecht, The Netherlands
{A.J.deMooij,d.dellanna,B.S.Logan,M.M.Dastani}@uu.nl

Abstract. Agent-based simulation is increasingly being used to model social phenomena involving large numbers of agents. However, existing agent-based simulation platforms severely limit the kinds of the social phenomena that can modeled, as they do not support large scale simulations involving agents with complex behaviors. In this paper, we present a scalable agent-based simulation framework that supports modeling of complex social phenomena. The framework integrates a new simulation platform that exploits distributed computer architectures, with an extension of a multi-agent programming technology that allows development of complex deliberative agents. To show the scalability of our framework, we briefly describe its application to the development of a model of the spread of COVID-19 involving complex deliberative agents in the US state of Virginia.

Keywords: Distributed simulation · Agent-based simulation · Social Simulation

1 Introduction

Social simulation [20] is increasingly being used to study complex social phenomena such as the evolution of economic inequality, environmental pollution, seasonal migrations, spreading of diseases, traffic, etc., and to train professionals such as police and fire brigades when confronted with incidents involving a large number of people. A key approach to studying such social phenomena is agent-based modeling and simulation. State-of-the-art agent-based simulation platforms are capable of supporting the synchronized execution of large numbers of agents by exploiting the computing power of distributed computer architectures such as computing grids. However, these platforms support only very simple agent behavior models, which severely limits the kinds of social phenomena that can be modeled [17, 25, 19]. On the other hand, existing multi-agent programming languages support the high-level social and cognitive concepts necessary to model the complex agent behaviors required for social simulations. However, these multi-agent programming languages and platforms are generally not designed to support the synchronized distributed execution of large numbers of agents.

In this paper, we present a novel simulation framework for the distributed simulation of large-scale multi-agent systems consisting of intelligent autonomous agents

that can perform complex tasks such as sensing, reasoning, and planning. To create this framework, we have developed a new discrete time distributed agent-based simulation platform called PanSim, and Sim-2APL, an extension to the 2APL Java-based multi-agent programming library that provides support for the development of agent-based simulations.[3] Sim-2APL supports the implementation of intelligent autonomous agents and multi-agent systems in terms of high-level social and cognitive concepts. PanSim provides scalability by distributing the execution of individual Sim-2APL agent programs over multiple computing resources in a synchronized manner in order to scale the execution of large-scale agent-based simulations. We present a synchronized execution model and state some minimal constraints on the use of Sim-2APL necessary to allow integration with PanSim and ensure the repeatability of simulations.

In order to demonstrate the applicability and scalability of the PanSim + Sim-2APL simulation framework, we report on experiments involving an agent-based simulation of the spread of COVID-19 in seven counties in the US state of Virginia. The input to the simulation consists of a synthetic population with realistic demographics, weekly activity schedules, and activity locations drawn from real location data. In the chosen counties, the number of individuals ranges from 20k to 180k and the number of weekly visits to locations ranges from 680k to about 6 million. Each individual in the synthetic population is represented by a Sim-2APL agent which reasons about whether to comply with non-pharmaceutical interventions such as mask wearing and social distancing that were introduced in Virginia between March and July 2020. In the current paper, we focus on the engineering of the PanSim + Sim-2APL framework, and we refer the reader to a companion paper for details of the simulation model [16].

Organization. The rest of the paper is organized as follows: In Section 2 we discusses related work on large-scale simulation with complex agent models. Section 3 and Section 4 present the design of PanSim and Sim-2APL respectively. Section 5 describes an exemplar simulation that simulates COVID-19 epidemic evolution jointly with a 2APL behavior model that we use to study the scaling properties of PanSim + Sim-2APL. Section 6 presents the results of the scaling experiments. Finally in Section 7 we end with concluding remarks.

2 Related Work

A number of platforms have been developed to address the challenges of scaling simulations. Notable successes have been obtained by exploiting domain semantics [4, 6], or by using simplified models of agents. In the context of epidemic simulations, for instance, agent behavior is often characterized only by a simple finite state machine which represents the progression of the disease. Bhatele et al. [5] were able to demonstrate an epidemic simulation scaling up to the size of the US population, which computed each simulated day in 57.8 ms on 655,360 cores of the Blue Waters supercomputer. The proposed simulator was heavily optimized for the particular application and architecture, and the agents did not model any complex cognitive behavior.

[3] Source code for PanSim is available at https://github.com/parantapa/pansim, and that for Sim-2APL is available at https://bitbucket.org/goldenagents/sim2apl

On the other hand, simulation platforms that support more complex agent models are typically designed for ease of development, maintenance, and *post hoc* analytics. For example, Barrett et al. [3] developed a large-scale disaster simulation with a database-centric simulation architecture where different modules compute various aspects of the simulation, such as transportation, communication, health states, behavioral choices, etc. The architecture allowed these modules to be separated and developed independently by multiple developers using different programming languages, data structures, and parallelization schemes, and to be plugged in and out as needed. The database-centric interaction between modules also results in all intermediate states being stored systematically, which facilitates debugging and later analysis. While this approach allows rapid development and complex representations of agents, there is a price to be paid in terms of scalability. The simulation needed over 16 hours to compute 100 time steps with $\sim$700,000 agents. Other approaches to scaling include dynamically varying the resolution of the simulation [23], and developing hybrid simulations that allow a mixture of simple and more complex agent models [26].

Simulating individual agents whose behaviors depend on their observations and internal states requires a decision making component that allows them to reason, decide and plan their actions. Various theories of decision-making have been proposed, from rational decision theories and BDI theory [14] to more psychologically-based approaches such as the Theory of Planned Behavior (TPB) [21]. These theories propose various conceptualizations of decision-making behavior in terms of motivational, informational and deontic attitudes, together with a decision rule that determines which of an agent's available actions will be selected based on the agent's attitudes [10]. To facilitate the development of autonomous agents based on these behavioral theories, a number of dedicated programming languages have been proposed where the agents' decisions are directed by their beliefs, goals, plans, and actions [7, 8]. For example, Bordini and Hübner [9] show how complex BDI agents programmed in Jason can be used for social simulation. In their approach, the agents' environment is implemented by extending a predefined Java class. Caballero *et al.* [11] also implement agents in Jason, but use the simulation platform Mason to simulate the environment. In both these approaches, the number of agents that can be simulated is limited by the number of threads available in the JVM. For a comprehensive survey of the use of BDI agents and complex reasoning in social simulations we refer the reader to the paper by Adam and Goudou [1]. Here, we note only that Adam and Gaudou identify scalability as a key issue limiting the use of BDI agents in simulations, and state that the distribution of a simulator over a network is "a very difficult problem that is far from being solved" [1, p. 228].

3 PanSim Design and Implementation

The current framework is quite broadly applicable to social contagion-like phenomena, such as the spread of behaviors, information, technologies, infectious diseases, etc. In this section, we describe how PanSim is structured to allow scalable computation of contagions through a population.

PanSim is a multi-contagion simulator, where two contagion processes progress concurrently on top of a dynamic contact network. In PanSim's design we assume that one of these contagion processes is a simple contagion, that is it can be fully described declaratively using a SIR like model [24]. PanSim provides its own configuration language to describe this simple contagion. On the other hand, very few assumptions are made about the nature of the other contagion process, which is assumed to be complex.[4] Authors of PanSim simulations are expected to provide custom code that encapsulates the progression and transmission logic for the complex contagion.

PanSim is a discrete time agent-based simulation framework. A simulation in PanSim progresses in discrete timesteps, and within a given timestep the simulation progresses in multiple sequential phases. However, within any given phase computations corresponding to different agents progresses concurrently.

The dynamic contact network in PanSim is specified in terms of a temporal agent-location bipartite graph. In PanSim agents interact with each other at specific locations. The locations visited by a given agent can change from one timestep to the next. Agents that are at the same location at the same time come into contact with each other. The contact network of agents thus formed is the unipartite projection (on the agent set) of the dynamic bipartite agent-location network.

In the following, to make the presentation more concrete, we describe the implementation of a behavior-aware COVID-19 simulation as a running example. Full details of the simulation can be found in a companion paper [16]. In this scenario, a COVID-19 disease model serves as the simple contagion, while a Sim-2APL-based socio-psychological behavior model takes the role of the complex contagion. For the rest of this paper we use the terms disease model and simple contagion model interchangeably. Similarly, we also use the terms socio-psychological model and complex contagion model interchangeably.

An agent's state comprises of two parts, its disease state and its socio-psychological state. Further, behavior exhibited by the agents is categorized into two classes: a) disease modifier behaviors and b) visible attribute behaviors. *Disease modifier behaviors* — such as wearing masks, social distancing, etc. — modify disease transmission properties, while *visible attribute behaviors* — such as displaying religious or political affiliations or symptoms of the disease — are used to indicate the agent's stance and influence other agents.

In this model, during a given simulation timestep the following steps are executed. *First*, every agent in the system, based on their current socio-psychological state, 'decides' which locations to visit, as well as how to 'behave' during each of those visits. These behaviors include disease modifier behaviors, as well as visible attribute behaviors. *Second*, when visiting a location the agents come into contact with each other. During this step, disease transmission takes place from infectious to susceptible agents. Also, the agents interact with other agents and 'see' their visible attributes. *Finally*, for every individual agent their disease state progresses, and they update their socio-psychological state based on their current disease state as well as their observations of other agent's visible attributes that they came into contact with.

[4] Here we use the terms simple and complex contagions in their literal sense and not specifically in the sense developed and popularized in [13].

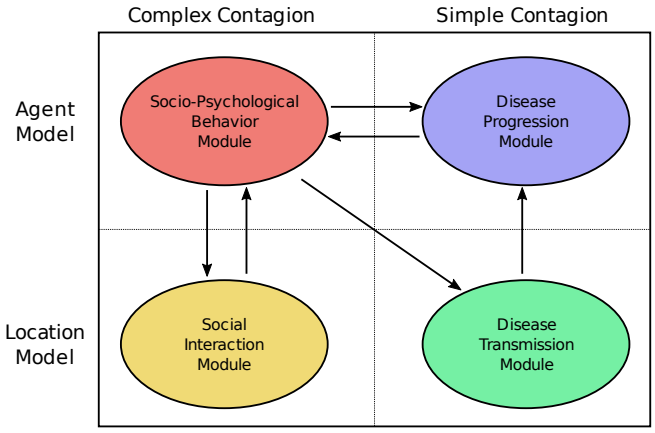


Fig. 1: Structure of a PanSim simulation

3.1 Structure of a PanSim Simulation

From the perspective of PanSim, the structure of a PanSim simulation consists of four major modules: the socio-psychological module, the social interaction module, the disease transmission module, and disease progression module. Figure 1 shows the overall organizations of the modules and their communication patterns.

The socio-psychological module and the social interaction module together represent the complex contagion component of a PanSim simulation, while the disease transmission and progression modules together represent the simple contagion component of the simulation. Another way of organizing the modules is to think of them from the perspective of the dynamic agent-location bipartite graph that serves as network on which both contagions progress. In this view, the socio-psychological and disease progression models encapsulate the computation that happens on behalf of every agent/individual in the simulation, while the social interaction and disease transmission modules encapsulate the computation that happens on behalf of every location in the system.

To write a custom PanSim simulation, the simulation authors only need to provide the code for the socio-psychological behavior module. The rest of modules are provided by PanSim itself. For example, in the exemplar problem described above, the socio-psychological module is written using the Java Sim-2APL library (described in Section 4). PanSim provides a generic language-agnostic interface, written using Apache Arrow[5], that can be used to write the socio-psychological module in most popular programming languages, including C, C++, Java, Python, and R.

3.2 Formal Description of a PanSim Simulation

Here we formally describe the structure of a PanSim simulation. A stochastic discrete time simulation can be written as a stochastic function $F : S \rightarrow S$ that, given the state

[5] https://arrow.apache.org/

of a system at timestep t, $s^t \in S$, computes the next state of the system $s^{t+1} = F(s^t)$. The whole simulation can then be formulated as an iterated application of the simulation function F, starting from the initial state s^0.

To use distributed system hardware, it is important to split this monolithic function into parts that can be executed in parallel, with intermediate coordination. For this purpose we consider the following decomposition of the system state at timestep t, $s^t = (s_1^t, s_2^t, \ldots, s_n^t)$. Here, s_i^t is the state of the ith agent at time t. As described above, PanSim implements a two contagion model, that we refer to as the socio-psychological model and the disease model. The state of the agent then is split as $s_i^t = (b_i^t, d_i^t)$ where b_i^t and d_i^t are the state of the agent corresponding to the socio-psychological and disease models. Equations Eq. 1–5 show the functional decomposition that is used in PanSim to compute the next state of an agent given the current state.

$$(L_i^t, \tau_i^t, v_i^t, m_i^t) = \nu(b_i^t, d_i^t) \tag{1}$$

$$\Delta b_i^t = \sum_{j;j\neq i} \sum_{l \in L_i^t \cap L_j^t} \beta\left(v_i^t(l), v_j^t(l), \tau_i^t(l), \tau_j^t(l), l\right) \tag{2}$$

$$\Delta d_i^t = \sum_{j;j\neq i} \sum_{l \in L_i^t \cap L_j^t} \rho\left(d_i^t, d_j^t, \tau_i^t(l), \tau_j^t(l), m_i^t(l), m_j^t(l), l\right) \tag{3}$$

$$d_i^{t+1} = \sigma(d_i^t, \Delta d_i^t) \tag{4}$$

$$b_i^{t+1} = \gamma(b_i^t, \Delta b_i^t, d^{t+1}) \tag{5}$$

First, as part of the socio-psychological model, the locations to be visited by the agent (L_i^t), the time durations of the visits (τ_i^t), the visual attributes displayed by the agent during the visits (v_i^t), and the disease modifier behaviors observed by the agent (m_i^t) are computed. In the formal model the socio-psychological model is represented by the function $\nu()$ (Eq. 1). Second, for each pair of agents visiting the same location, the social interaction updates, Δb_i^t, and disease transmission updates, Δd_i^t, are computed using the interaction model $\beta()$ (Eq. 2), and disease transmission model $\rho()$ (Eq. 3) respectively. Third, agent's disease state is updated to d_i^{t+1} using the disease progression model $\sigma()$, based on their current disease state d_i^t and disease transmission update Δd_i^t (Eq. 4). Finally, the agents socio-psychological state is updated to b_i^{t+1} using the socio-psychological update model γ, based on their current socio-psychological state b_i^t, their interaction updates Δb_i^t, as well as their updated disease state d_i^{t+1} (Eq. 5).

Note, we intentionally do not describe the domain of the state and update variables, b_i^t, Δb_i^t, etc. They can be modeled using a variety of structures that support the required operations. In the experiments shown below, they are implemented using real-valued vectors of appropriate lengths.

3.3 Declarative Simple Contagion Model

In PanSim the simple contagion's definition is written in a TOML[6]-based domain-specific language. Table 1 shows the simple contagion model (a COVID-19 disease

[6] https://github.com/toml-lang/toml

Table 1: Simple contagion model (Covid-19 disease model)

Category	Parameter	Value
	unit_time	300.0
	states	[succ, expo, isymp, iasymp, recov]
	behaviors	[base, mask, sdist, mask_sdist]
	exposed_state	expo
susceptibility	succ	1
infectivity	isymp	4.81e-05
	iasymp	2.40e-05
progression	expo	{isymp = 0.6, iasymp = 0.4}
	isymp	{recov = 1.0}
	iasymp	{recov = 1.0}
dwell time	expo	{isymp = dist1, iasymp = dist1}
	isymp	{recov = dist2}
	iasymp	{recov = dist2}
distribution	dist1	{dist = fixed, value = 6}
	dist2	{dist = fixed value = 14}
behavior modifier	base	{base = 1.0, mask = 0.5, sdist = 0.5, mask_sdist = 0.25}
	mask	{base = 0.5, mask = 0.25, sdist = 0.25, mask_sdist = 0.15625}
	sdist	{base = 0.5, mask = 0.25, sdist = 0.25, mask_sdist = 0.15625}
	mask_sdist	{base = 0.25, mask = 0.15625, sdist = 0.15625, mask_sdist = 3.906e-3}

model) used for the scaling studies described later in this paper. At its core the model is a SEIAR model with five disease states: susceptible (succ), exposed (expo), infected symptomatic (isymp), infected asymptomatic (iasymp) and recovered (recov).

Disease transmission happens when a susceptible individual (susceptibility > 0) comes in contact with an infectious individual (infectivity > 0). The probability of transmission is defined in terms of unit interaction times, specified in the configuration in seconds. If an individual with susceptibility α comes in contact with an individual with infectivity β, for unit time, then the probability of disease transmission is given by $\alpha \times \beta$. In the given example (Table 1), if a susceptible individual is in contact with an infectious (symptomatic) individual for 300 seconds, and both have baseline behaviors, then the probability of the susceptible individual getting infected is 4.81×10^{-5}.

Disease transmission probability is further affected by disease modifier behaviors. For example in the configuration shown in Table 1, four disease modifier behaviors are defined: baseline (base), wearing masks (mask), social distancing (sdist) and wearing masks as well as social distancing (mask_sdist). If in the above example a susceptible individual wearing masks interacts with an infectious (symptomatic) individual wearing masks and social distancing, for 300 seconds, then the probability of disease transmission for this case is given by 7.51×10^{-6}.

An individual with a given disease state may move to a different disease state, based on the progression of the disease inside the individual. In the given example, three progressions are defined. An individual in the exposed state will move to one of infectious states. Further there is a time — measured in simulation timesteps, specified in days — after which the progression to a different state occurs. In the given example (Table 1), infected individuals move to recovered state after 14 simulation timesteps (or 14 days).

Some of the parameters used fix values in the model come from COVID-19-related information shared by public health agencies, such as CDC [12]. The rest of the param-

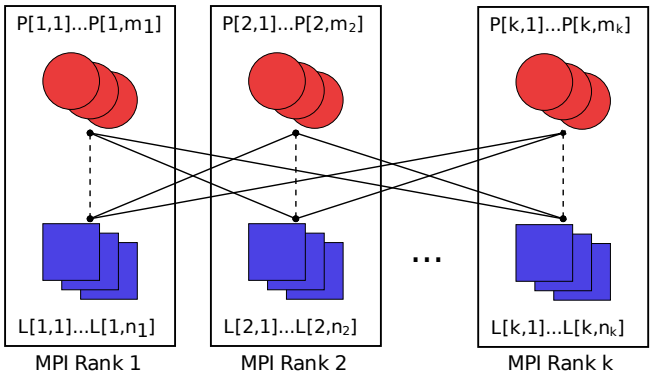


Fig. 2: Partitioning of agents/individuals and locations for distributed processing on PanSim.

eters are obtained by calibration to data. The procedure for calibration and details about how the model was arrived at can be found in our companion paper [16].

3.4 Distributed Software System Implementation

PanSim is a MPI based distributed memory application that is implemented in a mix of Python and C++. In PanSim a Python/C++ process (MPI rank) runs on each CPU core available. If the socio-psychological module is not written in Python, as is the case for the current study, then the socio-psychological module is run as a separate process. The socio-psychological processes share the CPU cores with the PanSim processes[7]. In this scenario, data is shared between PanSim processes and the socio-psychological module processes using Apache Arrow specifications.

On PanSim the two contagions progress over a dynamic agent-location bipartite graph. To be able to utilize distributed computing hardware, the nodes in the agent-location bipartite graph are partitioned across the MPI ranks. Figure 2 shows the overall partitioning strategy. To partition the graph evenly across the MPI ranks while keeping the cross-rank edges at a minimum, we use a two-step greedy process. In the first stage, the locations in the bipartite graph are sorted based on their maximum indegree. Next, the locations are assigned to the MPI ranks in a round robin manner. Finally the agents are assigned to the rank of the location that they are likely to visit the most frequently, which in most cases is their home location[8].

PanSim uses a bulk synchronous parallel design [18]. A PanSim simulation progresses in discrete timesteps. Within a timestep the execution progresses in five distinct

[7] To ensure that the socio-psychological module processes and PanSim processes don't compete for CPU resources we use MPI implementation specific configuration to make PanSim processes sleep during the execution of the socio-psychological module. This configuration trades of some performance for ease of programming.

[8] We experimented with using Metis and ParMetis [22] for this partitioning. However, we found that our simple approach was much faster and produced adequately good partitions.

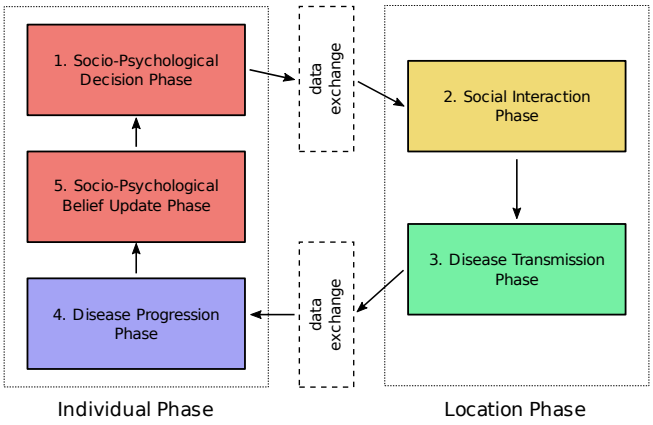


Fig. 3: Different phases of computation in a single timestep of a PanSim simulation.

phases, as described formally in Section 3.2. Figure 3 shows the different phases of computation of a PanSim simulation for a single timestep. First, in the socio-psychological decision phase (Eq. 1), every agent decides the locations to visit, and how to behave during those visits. This is followed by data exchange among MPI ranks to transfer information to the rank corresponding to the location of the visits. Second, in the social interaction phase (Eq. 2), the interactions of the individuals at a every location is computed. Third, in the disease transmission phase (Eq. 3), the probability of susceptible agents getting infected from visits is computed. After the third phase, data is again exchanged among the MPI ranks to send the social interaction and transmission updates back to the agents they correspond to. Fourth, in the disease transmission phase (Eq. 4), the disease state of the agent is updated based on the transmission and progression models. Finally, in the socio-psychological belief update phase (Eq. 5), the socio-psychological agent state is updated based on the social interaction and the updated disease state of the agent.

As shown in Figure 3, the first, fourth, and fifth phases of the simulation are collectively referred to as the individual phases. The computation of these phases can progress concurrently for every agent. Similarly, the second and third phases are location specific and can be executed concurrently for every location.

4 Sim-2APL

Sim-2APL is an extension of the agent programming library Java-2APL [15] (2APL) which supports the development of complex reasoning agents for large-scale simulations. 2APL defines the concepts of beliefs, goals, plans, and reasoning rules as Java interfaces, and dictates the interaction between these interfaces. In 2APL, the *Context* captures the agent's information or beliefs, the *Triggers* capture events or goals the agent may react to, the *Plans* capture specific parts of behavior that agents can perform, and the *Plan Schemes* match triggers to a suitable plan to be executed. An agent's behavior is generated through the application of *plan schemes* to *triggers*. The execution of an

agent is defined in terms of pre-programmed execution steps, which are captured in the agent's *deliberation cycle*. The steps in the deliberation cycle allow plan schemes to be applied in response to different types of triggers (see [15] for more details on 2APL).

In agent-based simulations, agents sense the environment and act upon it. From the point of view of Sim-2APL, PanSim acts as the environment in which agents sense and act. However, to allow the agents to effectively act and interact in the environment, the action execution and deliberation cycle of 2APL must be modified. In 2APL, the deliberation cycle of an agent is rescheduled as soon as it ends. This means agents are executed continuously and independently, and an agent does not have to wait for all other agents to finish their deliberation cycle before acting. As a result, one agent may perform several deliberation cycles – and thus act in the environment several times – while another agent is still computing its first deliberation cycle. While this approach is appropriate for many applications, it does not guarantee a synchronized execution of the agents, which in turn may make simulations not repeatable. To address this, we modified 2APL in two ways: first, the execution of agent actions is delegated to the environment; second, we constrain the way agents are scheduled and executed.

Action execution In 2APL, the external actions in a plan are executed directly through Java method calls. This means that agents have full control over *when* actions are executed. However, many simulation platforms, including PanSim, do not allow agents to change the state of the environment directly, but rather update the state of the environment by calculating the subsequent simulation state from the joint set of all agent actions. For example, in PanSim this is represented by the stochastic function F in Sec. 3.2. Therefore, in the framework, the execution of external actions is delegated to the environment. In addition, we require that each plan executes at most one external action per deliberation cycle.[9] This is achieved by modifying the 2APL `Plan` interface so that its `execute()` method (`void` in 2APL) returns to the environment an identifier for the intended action that otherwise would be performed directly through a method call. When actions are delegated to the PanSim environment, the identifier is the tuple $(L_i^t, \tau_i^t, \epsilon_i^t, m_i^t)$ from Eq. 1.

Agent scheduling As explained above, many simulation platforms require agent execution to be synchronized to discrete-time steps. In Sim-2APL, discrete-time synchronization of agents is achieved using three interfaces: `StepExecutor`, `StepGenerator`, and `EnvironmentInterface`, the interaction of which is visualized in Fig. 4. The `StepExecutor` interface defines the method `doStep`, which is responsible for making each agent perform a single time step (deliberation cycle), and a method `reschedule`, called by each agent to reschedule its deliberation cycle for the subsequent time step. The `StepGenerator` interface specifies how execution time alternates between the `StepExecutor` and the environment responsible for storing and advancing the simulation state. This interface waits for the environment to finish calculating the new state at each time step. Finally, the `EnvironmentInterface` implements the communi-

[9] Since agents can execute multiple plans during one deliberation cycle, this approach does not restrict the agent's number of actions per time step.

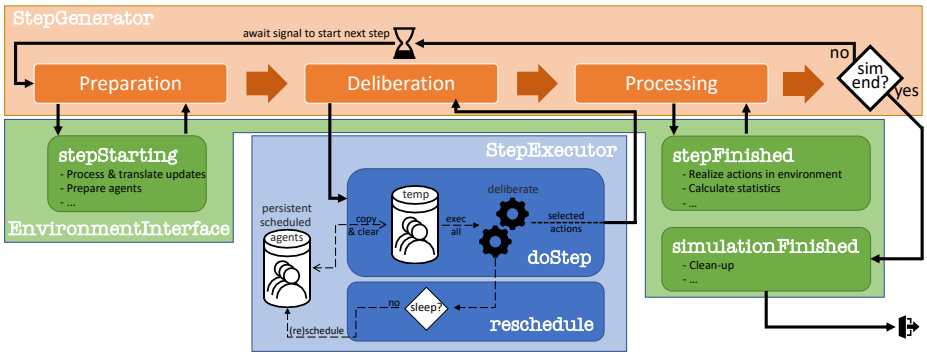


Fig. 4: The `StepGenerator` calls the appropriate methods on the `StepExecutor` and registered `EnvironmentInterfaces` to initiate the *preparation*, *deliberation*, and *processing* phases in each time step.

cation layer with the environment. In the following, we describe these three interfaces in more detail.

StepGenerator The `StepGenerator` is responsible for initiating the next time step in Sim-2APL. Each step is divided into three phases: *preparation*, *deliberation*, and *processing*, each of which run on the main thread so that the next phase only starts when the previous phase has finished. The process of phase transitions in the `StepGenerator` is visualized at the top in Fig. 4. The `StepGenerator` interface does not specify when a new step starts as this is initiated by an external 'driver'; in the framework, PanSim signals the `StepGenerator` to begin the next time step. This ensures agents cannot deliberate while the state of the environment is being updated. During the *preparation* phase, the `stepStarting` method of the `EnvironmentInterface` is called to prepare for the next phase of agent deliberation. The `stepStarting` method should perform all computations necessary to prepare for the agents' deliberation at this time step, such as processing or translating updates from the environment, creating *Trigger*s for belief updates, or calculating global resources or statistics. When the *preparation* phase is complete, the *deliberation* phase for this time step is started by calling the `doStep` method of the `StepExecutor`. When deliberation of all agents is completed, the `StepExecutor` returns the actions generated by the agents. These actions can then be ordered to ensure determinism (e.g., using agent names) and are passed to the `EnvironmentInterface` to start the final *processing* phase. In this phase, the environment realizes the effect of the actions generated by the agents and calculates the next simulation state.

EnvironmentInterface Sim-2APL is agnostic about what environment it is connected to. In order for Sim-2APL to interact with an environment, the `EnvironmentInterface` must be implemented. This interface is responsible for encoding agent actions and sending them to the environment, and receiving state updates from the environment and translating those for use by the agents. The interface defines three methods: `stepStarting`

and `stepFinished`, which are called during the *preparation* and *processing* phases of the `StepGenerator`, respectively, and `simulationFinished` which is called when the simulation ends. This interface and its methods are shown in green in Fig. 4. Implementation of the `stepStarting` method is optional. The `stepFinished` method receives the set of actions produced by the agents as an argument, and should realize the actions in the environment and produce the next simulation state. In the framework, this is achieved by sending all agent actions to PanSim. However, in a simulation where the environment is programmed in Java, one could use the same methods that in 2APL are called on directly by the agents to realize the effect of those actions. Finally, the `simulationFinished` method is called when the simulation has ended. This method should implement any necessary cleanup operations, such as closing the connection with the environment. Multiple `EnvironmentInterface`s can register with the `StepGenerator` by calling its `registerEnvironmentListener` method. The appropriate methods of each `EnvironmentInterface` instance will be called sequentially in each of the three phases. Note that the only assumptions we make regarding the environment are that, (i) there is *some* way for it to interface with Java so that actions can be executed in it and the state can be requested by agents, and (ii) the simulation state can be advanced one step at the time.

Step Executor The `StepExecutor` is responsible for executing a single deliberation cycle for each agent at the current time step. In our *default implementation*, the `StepExecutor` maintains a queue of the deliberation cycles of scheduled agents. As in 2APL, agents re-schedule themselves from within their own deliberation cycle (unless they sleep). To ensure an agent is not executed twice within the same time step, when the `doStep` method of the `StepExecutor` is called by the `StepGenerator`, the queue is first copied into a temporary queue. The deliberation cycles of all agents are executed from this temporary queue, and rescheduled agents are placed on the original, now empty, queue. This process is visualized in blue in Fig. 4. Execution is handled using a Java Executor service, allowing the deliberation phase to run concurrently. Agents' (external) actions are then collected from the deliberation cycles and placed into a hash map where the unique identifier of the agent producing those actions is the key, and the value is the list of produced actions. This hash map is then returned to the `StepGenerator`.

5 Sample Simulation

We now describe our simulation of the spread of COVID-19, instantiated using PanSim + Sim-2APL. The simulation is built using a synthetic population of several counties in the US state of Virginia. Agents are represented with detailed demographic information from the US Census Bureau, along with detailed weekly activity sequences, and appropriate locations assigned for the activities from comprehensive location data [2]. The disease spread is driven by the interactions between agents (due to physical collocation), as they go about their weekly activity schedules. In order to model changes in activity patterns as various social distancing interventions were instituted, we developed a normative reasoning model for the agents using Sim-2APL, as described below. The

Table 2: The counties of the state of Virginia used for the experiments, along with the number of persons, households, and weekly location visits in the synthetic population.

County	Persons	Households	Visits
Goochland	20,923	8,240	680,571
Fluvanna	24,110	9,776	779,337
Louisa	32,938	13,398	1,066,179
Charlottesville	41,120	18,377	1,335,596
Albemarle	93,570	39,920	3,047,807
Hanover	98,435	38,149	3,204,317
Richmond	181,975	89,146	5,920,569

simulation was calibrated using both COVID-19 case data and cellphone-based mobility data. Since this was an extensive research project in itself, we have presented that aspect of the work in another paper [16]. In this paper, we have focused on the engineering aspects of the simulation and present scaling experiments below in Section 6.

The simulation proceeds as follows. On each simulated day, each agent chooses which of its activities from its normal (pre-COVID) schedule it will carry out. The deliberation process is informed by normative reasoning as we describe below. For the activities the agent selects, it also chooses which behavioral interventions (mask-wearing, physical distancing) it will comply with, while carrying out each activity. Each activity results in a *visit* to a corresponding location. Table 2 shows the number of persons, households, and visits in each county, where the visit counts are based on pre-COVID activity schedules. During the simulation period, as the agents reduce their mobility to comply with various norms, the number of visits are lower.

For our sample simulation, we consider the behavioral interventions of official institutes as *norms* which agents can reason about. We classify these norms as either regimented (R) – meaning that an agent has no choice but to comply, or non-regimented (NR) – meaning an agent is expected to comply but has the agency to violate. Examples of R norms are closure of schools and businesses, while examples of NR norms are wearing a mask or staying home when sick. Both NR and R norms operate on goals g, and are implemented in terms of the functions $applies : g \mapsto \{true, false\}$ and $transform : g \mapsto g \times \bot$, the former specifies whether the norm n applies to the goal while the latter transforms the goal g into a goal g' that complies with the norm, or into $\bot$ to not pursue the goal for one deliberation cycle. NR norms specify one additional function $attitude : g \times a \mapsto x \in (0, 1) \subset \mathbb{R}$, which also takes the agent a as a parameter and, based on beliefs, observations and attitudes of the agent a, calculates its motivation to comply with norm n as a probability $p(n, g)$.

The normative reasoning process which we employ is as follows. If the plan scheme of the agent a is triggered by a goal g, all norms that apply to g are collected and iteratively applied to g by using the $transform(g)$ function. After this step, a plan is selected for the updated goal following the traditional 2APL approach.

In our work, we interpret the daily activities in the activity schedules of an agent directly as the agents' (to-do) goals. The transformations applied by the norms can

change the modality of these activities (i.e. wear a mask, maintain physical distance), change the time or duration of the activity, or cancel the activity for that day.

For each location, PanSim computes the duration of overlap for each pair of agents that visits that location on the current day. This duration, coupled with whether the agents are complying with mask-wearing and physical distancing, determines the probability of infection if one of the agents is infectious and the other is susceptible. Based on these probabilities, PanSim computes disease state changes for all the agents. These are communicated back to the agents, along with the observations made by the agents of the visible attributes of the other agents they encounter, as described in Section 3. Each agent then uses this information in its decision-making procedure for the next simulated day, using Sim-2APL. The computational burden of the simulation is thus divided between the two components. As discussed earlier, prior work has either ignored individual behavioral complexity in favor of scaling disease spread simulations, or has focused on creating complex simulations with smaller agent populations. Our goal is to be able to scale simulations with complex individual agents to large population sizes, so we turn to scalability experiments with PanSim+Sim-2APL next.

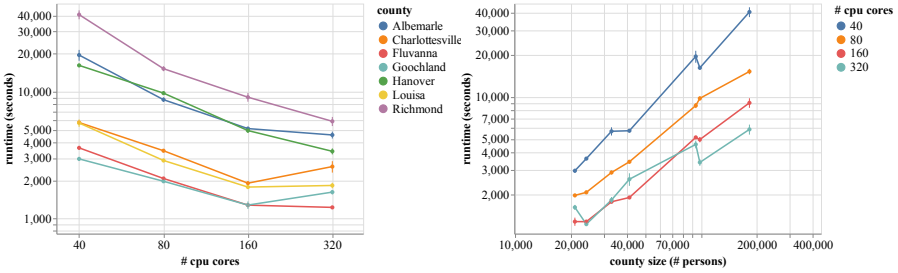
6 Scalability Experiments

For the purposes of the scaling experiments, we chose synthetic populations of seven counties in the state of Virginia, USA, with varying sizes. Table 2 shows the number of persons, households and their weekly activity schedule (location visits) in the synthetic populations. To understand the scalability of PanSim+Sim-2APL we ran individual simulations for each of the seven counties, with each simulation running for 180 timesteps representing 180 days starting from March 1, 2020. The simulations were run with 40, 80, 160, and 320 CPU cores on compute nodes each having 2 Intel Xeon Gold 6148 CPUs with 20 CPU cores each. The compute nodes used to run the experiments also had 384 GB of DDR4 RAM Memory and were connected to each other with Mellanox ConnectX-5 network adaptors. Each simulation was run 10 times and their running time was noted.

We study scaling in two ways. First, we keep the problem size fixed and increase the number of CPU cores. This is done by running the simulation for each county with the four levels of cores above. The expectation is that the running time should decrease smoothly as the computational resources increase.

Second, we keep the computational resources fixed and increase the problem size. This is done by comparing the running times for simulations of increasingly larger counties, while keeping the number of CPU cores fixed. We carried out this experiment for all four levels of CPU cores also. The expectation is that the running time should not increase too sharply as the problem size increases.

In both cases, the resulting performance curves should ideally be linear. However, communication overheads can make the curves nonlinear. There is also inherent nonlinearity in the structure of the problem, as the disease spread computation is quadratic in the number of agents simultaneously present at a location. It is also expected that at some point, the overhead of communication between distributed parts of the simulation becomes higher than the efficiency gained by splitting the computation across multiple



(a) (b)

Fig. 5: The mean runtime of PanSim+Sim-2APL simulations for seven counties of the state of Virginia compared with (a) the number of cores, and (b) the number of agents.

cores. For smaller problem sizes (i.e., smaller counties), this should become apparent with fewer cores.

6.1 Results

Figure 5 shows the variance in the runtime of the simulations when run with different number of CPU cores. We can see in Figure 5a when the same simulation is run with increasing number of CPU cores (strong scaling) for all the counties the runtimes decrease almost linearly till 160 CPU cores on a log-log scale. For smaller counties, such as Goochland and Charlottesville, increasing the number of CPU cores to 320 actually increases the runtime due the communication overhead becoming apparent, as discussed above. However, for a larger county like Richmond, the strong scaling results hold even with 320 CPU cores.

A similar story can be seen when looking at Figure 5b which shows the runtime of simulations with increasing compute load (number of agents in the county simulated). We can see that for counties with more than 100,000 persons, increasing the number of CPU cores to 320 shows definite benefits. However, for the rest of the counties simulated, the benefits of increasing CPU cores are observed only up to 160 CPU cores.

These results demonstrate that PanSim+Sim-2APL simulations integrate well, and can be used to simulate large populations. More detailed simulation results, investigating the effects of various non-pharmaceutical interventions, are presented in the companion paper, which focuses on the data, design, calibration, and analysis of the simulation [16].

7 Conclusion

In this paper, we presented a novel agent-based simulation framework for modeling large-scale complex social phenomena. We presented Sim-2APL, a Java-based multi-agent programming library that allows to model and simulate complex reasoning agents through the BDI paradigm. We integrated Sim-2APL with PanSim, our novel platform

for distributing large-scale agent-based simulations. We reported on a scalability experiment using a COVID-19 epidemic simulation with a population of BDI agents representing individuals from 7 counties of the state of Virginia, with population size ranging from 20k to 180k agents. Our results demonstrate that it is indeed possible to build an execute large-scale realistic simulations with BDI based agent models with efficient and judicious use of distributing computing platforms.

As we have seen with COVID-19 during 2020, epidemics (especially novel ones) are driven by human behavior. Until vaccines became available, public health authorities, institutions, and governments had to rely on non-pharmaceutical interventions to try to mitigate the epidemic. However, we don't have a rigorous understanding of the effectiveness of these interventions, due, in large part, to the complexity of human behavioral responses and their effects on epidemic dynamics. Thus, while there have been numerous computational and mathematical models of the COVID-19 epidemic that have been developed in the past year, they have largely focused on disease dynamics and have either ignored human behaviors or represented them in very simplistic ways, such as assuming that people comply with interventions independently with certain probabilities.

Our goal in developing this framework has been to bring together the strengths of MAS technologies for building normative reasoning agents with large-scale data-driven distributed agent-based simulation technologies. The scalability of this framework will now enable the development of more meaningful simulations, which can properly address complex human behaviors and allow reasoning about the effects of a larger class of interventions.

Acknowledgments

Parantapa Bhattacharya and Samarth Swarup were supported in part by NSF Expeditions in Computing Grant CCF-1918656 and DTRA subcontract/ARA S-D00189-15-TO-01-UVA.

References

1. Adam, C., Gaudou, B.: BDI agents in social simulations: a survey. Knowledge Engineering Review **vol. 31**(n° 3), pp. 207–238 (Jun 2016). https://doi.org/10.1017/S0269888916000096, https://hal.archives-ouvertes.fr/hal-01484960
2. Adiga, A., Agashe, A., Arifuzzaman, S., Barrett, C.L., Beckman, R.J., Bisset, K.R., Chen, J., Chungbaek, Y., Eubank, S.G., Gupta, S., Khan, M., Kuhlman, C.J., Lofgren, E., Lewis, B.L., Marathe, A., Marathe, M.V., Mortveit, H.S., Nordberg, E., Rivers, C., Stretz, P., Swarup, S., Wilson, A., Xie, D.: Generating a synthetic population of the United States. Tech. Rep. NDSSL 15-009, Network Dynamics and Simulation Science Laboratory (2015)
3. Barrett, C., Bisset, K., Chandan, S., Chen, J., Chungbaek, Y., Eubank, S., Evrenosoğlu, Y., Lewis, B., Lum, K., Marathe, A., Marathe, M., Mortveit, H., Parikh, N., Phadke, A., Reed, J., Rivers, C., Saha, S., Stretz, P., Swarup, S., Thorp, J., Vullikanti, A., Xie, D.: Planning and response in the aftermath of a large crisis: An agent-based informatics framework. In: Pasupathy, R., Kim, S.H., Tolk, A., Hill, R., Kuhl, M.E. (eds.) Proc. of the 2013 Winter Simulation Conference. pp. 1515–1526. IEEE Press, Piscataway, NJ, USA (2013)

4. Barrett, C.L., Bisset, K.R., Eubank, S.G., Feng, X., Marathe, M.V.: Episimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In: Proc. of the 2008 ACM/IEEE Conference on Supercomputing. pp. 37:1–37:12 (2008)
5. Bhatele, A., Yeom, J.S., Jain, N., Kuhlman, C.J., Livnat, Y., Bisset, K.R., Kale, L.V., Marathe, M.V.: Massively parallel simulations of spread of infectious diseases over realistic social networks. In: Proc. of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID. IEEE (2017)
6. Bisset, K., Chen, J., Feng, X., Vullikanti, A., Marathe, M.: EpiFast: A fast algorithm for large-scale realistic epidemic simulations on distributed memory systems. In: Proc. of the 23rd International Conference on Supercomputing (2009)
7. Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.): Multi-Agent Programming: Languages, Platforms and Applications, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15. Springer (2005)
8. Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.): Multi-Agent Programming: Languages, Tools and Applications. Springer (2009)
9. Bordini, R.H., Hübner, J.F.: Agent-based simulation using BDI programming in Jason. Multi-agent systems: simulation and applications pp. 451–471 (2009)
10. Broersen, J., Dastani, M., Hulstijn, J., Huang, Z., van der Torre, L.: The BOID architecture: Conflicts between beliefs, obligations, intentions and desires. In: Proc. of the 5th International Conference on Autonomous Agents. pp. 9–16 (2001)
11. Caballero, A., Botía, J., Gómez-Skarmeta, A.: Using cognitive agents in social simulations. Engineering Applications of Artificial Intelligence **24**(7), 1098–1109 (2011)
12. Centers for Disease Control and Prevention: COVID-19 pandemic planning scenarios (Accessed: 2020-10-06), https://www.cdc.gov/coronavirus/2019-ncov/hcp/planning-scenarios.html
13. Centola, D., Macy, M.: Complex contagions and the weakness of long ties. American Journal of Sociology **113**(3), 702–734 (2007)
14. Dastani, M., Hulstijn, J., van der Torre, L.: How to decide what to do? European Journal of Operational Research **160**(3), 762–784 (2005), Decision Analysis and Artificial Intelligence
15. Dastani, M., Testerink, B.: Design patterns for multi-agent programming. International Journal of Agent-Oriented Software Engineering **5**(2/3), 167–202 (2016)
16. de Mooij, J., Dell'Anna, D., Bhattacharya, P., Dastani, M., Logan, B., Swarup, S.: Quantifying the effects of norms on COVID-19 cases using an agent-based simulation. In: Proceedings of the The 22nd International Workshop on Multi-Agent-Based Simulation (MABS) (2021)
17. Dignum, F., Dignum, V., Jonker, C.M.: Towards agents for policy making. In: David, N., Sichman, J.S. (eds.) Multi-Agent-Based Simulation IX. pp. 141–153. Springer-Verlag, Berlin, Heidelberg (2009)
18. Gerbessiotis, A.V., Valiant, L.G.: Direct bulk-synchronous parallel algorithms. Journal of Parallel and Distributed Computing **22**(2), 251–267 (1994)
19. Gilbert, N.: When does social simulation need cognitive models? In: Sun, R. (ed.) Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation, pp. 428–432. Cambridge University Press, Cambridge (2006)
20. Gilbert, N., Troitzsch, K.G.: Simulation for the social scientist. Open University Press (2006)
21. Glanz, K., Rimer, B.K., Viswanath, K.: Health behavior and health education: Theory, research, and practice. John Wiley & Sons (2008)
22. Karypis, G., Schloegel, K., Kumar, V.: Parmetis. Parallel graph partitioning and sparse matrix ordering library. Version **2** (2003)

23. Navarro, L., Flacher, F., Corruble, V.: Dynamic level of detail for large scale agent-based urban simulations. In: Tumer, Yolum, Sonenberg, Stone (eds.) Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems. pp. 701–708. Taipei, Taiwan (May 2-6 2011)
24. Satsuma, J., Willox, R., Ramani, A., Grammaticos, B., Carstea, A.: Extending the SIR epidemic model. Physica A: Statistical Mechanics and its Applications **336**(3-4), 369–375 (2004)
25. Silverman, B.G., Johns, M., Cornwell, J., O'Brien, K.: Human behavior models for agents in simulators and games: Part I: Enabling science with PMFserv. Presence: Teleoper. Virtual Environ. **15**(2), 139–162 (2006)
26. Singh, D., Padgham, L., Logan, B.: Integrating BDI agents with agent-based simulation platforms. Auton Agent Multi-Agent Syst **30**(6), 1050–1071 (2016). https://doi.org/10.1007/s10458-016-9332-x