





Parallel and Distributed Epirust: Towards Billion-Scale Agent-Based Epidemic Simulations

Sapana Kale¹^a, Shabbir Bawaji¹, Akshay Dewan¹, Meenakshi Dhanani¹, Kritika Gupta¹,
Harshal Hayatnagarkar¹^b, Swapnil Khandekar¹, Jayanta Kshirsagar¹^c, Gautam Menon²^d
and Saurabh Mookherjee¹

¹Engineering for Research (e4r), Thoughtworks Technologies, Yerwada, Pune, 411 006, India

²Ashoka University, Sonapat, Haryana, 131029, India

Keywords: Epidemiology, Epidemic Simulations, Agent-Based Simulations, Large-Scale Simulations, Agent-Based Modeling, Parallel Computing, Distributed Computing.


Abstract: EpiRust is an open source, large-scale agent-based epidemiological simulation framework developed using Rust language. It has been developed with three key factors in mind namely 1. Robustness, 2. Flexibility, and 3. Performance. We could demonstrate EpiRust scaling up to a few millions of agents, for example a COVID-19 infection spreading through Pune city with its 3.2 million population. Our goal is to simulate larger cities like Mumbai (with 12 million population) first, and then entire India with its 1+ billion population. However, the current implementation is not scalable for this purpose, since it has a well-tuned serial implementation at its core. In this paper, we share our ongoing journey of developing it as a highly scalable cloud ready parallel and distributed implementation to simulate up to 100 million agents. We demonstrate performance improvement for Pune and Mumbai cities with 3.2 and 12 million populations respectively. In addition, we discuss challenges in simulating 100 million agents.


1 INTRODUCTION


EpiRust¹ is an open source, large-scale agent-based epidemic simulator written in the Rust programming language. It has been developed to balance three key factors for epidemic simulations namely 1. Robustness 2. Flexibility, and 3. Performance. Rust offers memory safety of managed run-times without their overheads (Matsakis and Klock, 2014).


The earlier EpiRust version was developed as a serial implementation using only single CPU core. The implementation was sufficient to simulate a large city like Pune with approximately 3 million population. However, it was not enough for running simulations with larger agent populations, such as for cities like Mumbai with more than 10 million population. We found that EpiRust runs slower for larger populations following a linear characteristic in a log-

log scale as lower performance caused by total higher lookups (see Figure 2). Thus, it was imperative to develop concurrent implementations to take advantage of multicore processors and multiple networked computers. Here, the Rust language offered benefits via the concurrency constructs either built into it or in the ecosystem. In the rest of this paper, we describe our journey of implementing parallel and distributed implementations of EpiRust. In the next section, we discuss the related work in this ecosystem, followed by the EpiRust model. Thereafter we explain our parallel and distributed implementations covering architecture, design, and cloud-readiness. Later, we share the details about large-scale experiments which we have run for Mumbai and Pune with their representative populations and discuss the performance numbers. Finally, we conclude with their limitations and our plans.

^a <https://orcid.org/0000-0002-7582-865X>

^b <https://orcid.org/0000-0002-2300-5539>

^c <https://orcid.org/0000-0002-0581-7218>

^d <https://orcid.org/0000-0001-5528-4002>

¹<https://github.com/thoughtworks/epirust/>

2 PAST WORK

Traditionally, epidemiologists relied upon calculus-based models for their simplicity and lower resources required for simulations. However, these models suffer with many significant downsides, such as aggregating dynamics at the population level, ignoring individuals' decision making, and assumption of homogeneous population (Kermack and McKendrick, 1927).

To rescue from this situation, agent-based models can simulate bottom-up social dynamics (Gilbert and Terna, 2000) such as epidemic spread (Patlolla et al., 2004). Here, an agent represents a person in a virtual society, and collectively their synthetic population is a simplified representation of the real target population with their demographic heterogeneity (Chapuis et al., 2022). This heterogeneity plays a crucial role in modeling of agent interactions and decisions (Dias et al., 2013; Wildt, 2015; Klabunde and Willekens, 2016; Hessary and Hadzikadic, 2017). In addition to this heterogeneity, the population size is another factor. The insights vary for different population sizes, and hence it is imperative to run the simulations as close to the real population size as possible (Jaffry and Treur, 2008; Kagho et al., 2022).

To develop agent-based epidemic simulations, one could consider from the available frameworks or libraries including Repast (Collier, 2003), NetLogo (Tisue and Wilensky, 2004), MASON (Luke et al., 2005), EpiFast (Bisset et al., 2009), GAMA (Tailandier et al., 2010), GSAM (Parker and Epstein, 2011), D-MASON (Cordasco et al., 2013), and OpenABL (Cosenza et al., 2018a).

A survey of agent-based simulation software (Abar et al., 2017) found that it is harder to develop and run models for extreme-scale simulations for it requires a huge amount of compute resources and their management.

One approach is to develop simplified models. For example, GSAM (Global Scale Agent Model) which observes that “*to track a contagion, simulating everyone's entire day-to-day schedule may not be necessary*” (Parker and Epstein, 2011). The alternative approach is to leverage the computer hardware and software solutions for scaling without sacrificing details (Parker and Epstein, 2011; Parry and Bithell, 2012). A few frameworks emphasize on addressing the needs for developing large-scale models, mainly because such models are inherently detail-oriented, complex, and resource-intensive (Eubank et al., 2004; Bisset et al., 2009; Parker and Epstein, 2011; Abar et al., 2017; Antelmi et al., 2019; Kshirsagar et al., 2021; Kerr et al., 2021). This pursuit requires over-

coming several challenges in improving scalability by combining proven and novel techniques.

A common technique is to split a large simulation model into multiple models and then to simulate them using multiple compute instances. This approach needs to take care of initialization, communication, and coordination as required. All these aspects are studied in a sub-discipline of simulation engineering namely *Co-simulation* (Gomes et al., 2018).

With multicore CPUs and GPUs becoming mainstream, it is imperative to harness such resources as demonstrated by OpenABL for GPUs and FPGAs (Cosenza et al., 2018a; Xiao et al., 2020). In addition, some of these resources can be better managed via a cloud computing environment, and thus it is highly desired to have a framework which could be cloud ready. An example is *HASH Platform* (Hash, 2022), an open-source platform which allows users to specify a model and execute it on its own cloud infrastructure. However, scalability of OpenABL and HASH does not exceed a few million agents.

Choosing a programming language for development also plays an important role, as it directly affects resource utilization for large simulations. For example, the work (Pereira et al., 2017) ranks programming languages for their use of CPU, memory, and energy consumption across different algorithms. In the context of agent-based simulations, these insights are corroborated in the work (Antelmi et al., 2019) which compares performance of Java-based MASON against a Rust-based framework for up to a million agents. The comparison confirms that Rust performs better for large-scale simulations consuming lower resources like CPU and memory.

Thus, it is imperative to combine these techniques together to achieve necessary performance and scalability for large-scale simulations. EpiRust is an attempt in this direction to develop an open-source performant, flexible, and most importantly robust agent-based simulation framework which could scale up to a billion agents towards mimicking the population of entire India.

3 EpiRust MODEL

For simulating epidemics, EpiRust follows a minimalist approach based on cellular automata (Gilbert and Terna, 2000). It models a virtual city with primary aspects of geography, heterogeneous population of agents, and disease dynamics. Geography is modeled as a grid which is divided into functional areas namely residential, transport, work, and hospital. Agents occupy these functional areas during the sim-

ulation according to their attributes and schedules.

The agent heterogeneity emerges from their demographic and professional attributes such as status of employment, nature of work (essential or non-essential), and preferred mode of transport (between public and private transport). Each agent lives in a dedicated home, and if employed, works in a work area. Agents move across different areas based on their respective schedules.

During these movements, agents come in close contact with each other spreading infection with certain probability. To evaluate this infection spread, the EpiRust model uses *Moore neighborhood* for eight neighboring cells (White et al., 2007). The disease dynamics is implemented as per the Mordecai SEIR compartmental model (Childs et al., 2020), which captures various disease states such as catching an infection, being infected, recovering and so on. As per the paper, we have specialized the model for COVID-19 (see figures 1a and 1b) as per the paper (Snehal Shekatkar et al., 2020).

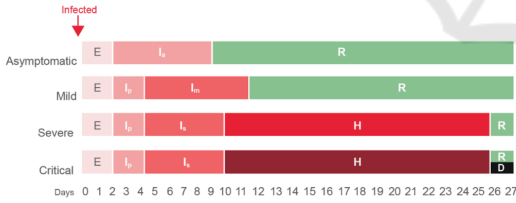
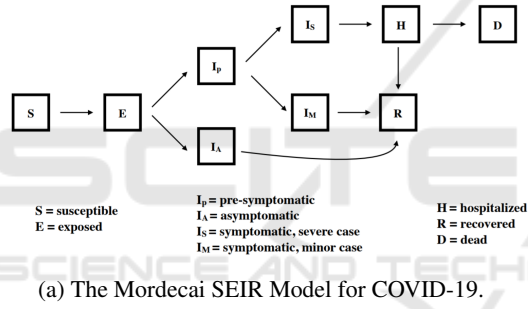


Figure 1: Disease Dynamics of COVID-19 (Snehal Shekatkar et al., 2020).

Compartmental models in epidemiology represent disease propagation via compartments like susceptible (S), exposed (E), infectious (I), and removed (R); removed translates to recovered or dead. These can be modeled mathematically or using agent-based models (for more details on mathematical models for epidemiology, please refer to the work (Brauer, 2008)). EpiRust can be configured to simulate general compartmental models like SEIR and SIR.

To contain a rapidly spreading infection in a simulation run, EpiRust supports three intervention strate-

gies namely *lockdown*, *isolation via hospitalization*, and *vaccination*. These intervention strategies can be configured before running the simulation.

A simulation progresses in discrete time steps (or *ticks*), and typically each step is mapped to an hour. Thus, twenty-four steps represent a day in the simulation. For each time step, the simulation iterates over all the agents, and computes their next disease state as per the disease dynamics model. This is the core algorithm for executing simulations which are depicted in Algorithm 1. The outer loop iterates over discrete time steps, whereas the inner loop iterates over agents. Thus, the set of rules and behaviors are executed for every agent during each time step. However, the order in which these behaviors are executed, and their internal states are updated, significantly alters the outcome of the overall simulation. It happens because an agent could refer to other agents' states such as their location on the grid which are no more original rather updated during their evaluation. This problem is known as *Path Dependence* (Gulyás, 2005). Path dependence is not desired within a time-step for its side-effects, but is desired across time steps for that it enables emergence in simulations.

Algorithm 1: Simplified Simulation Loop (Iterative Map-Reduce).

```

for each step in 1..n do
  if can_intervene then
    Apply intervention;
  end
  for each agent do
    Move agent on the grid;
    Update infection state;
  end
  if number_of_infected == 0 then
    Stop simulation;
  end
end

```

To counter this problem, EpiRust uses a double-buffering technique (Gulyás, 2005; Cosenza et al., 2018b; de Aledo Marugán et al., 2018) which maintains two separate buffers of agents' states for referring and updating each. These two buffers are swapped after each time step. Solving this problem also helps in writing a parallel version of the agents' loop which is described further in the following section along with the distributed architecture and implementation.

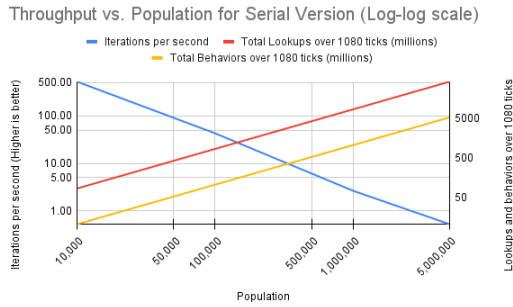


Figure 2: Throughput vs. Population for Serial Version (Log-log scale).

4 PARALLEL IMPLEMENTATION

EpiRust has a goal of simulating over a billion agents closer to the population of India. However, the serial, single-threaded implementation could not scale beyond 10 million agents and therefore the performance was sub-linearly slower for larger populations (see Figure 2 showing linear characteristics in a log-log scale). Thus, it was imperative to explore opportunities to harness modern multi-core processors, clusters of computers, and cloud environments via parallel and distributed implementation. Rest of this section describes approaches for parallel implementations whereas the following section continues for the distributed implementation.

To harness multi-core processors, we needed data and/or task parallelism opportunities. Agent iterations in Algorithm 1 offer data parallelism. A couple of caveats are around path dependency and collisions in agent placements. As discussed in the earlier section, the solution based on double-buffering addresses the path dependency problem enabling parallel evaluation of agent states except in case of collisions in agent placements. The agent collisions impose synchronization penalties which we decided to ignore for now.

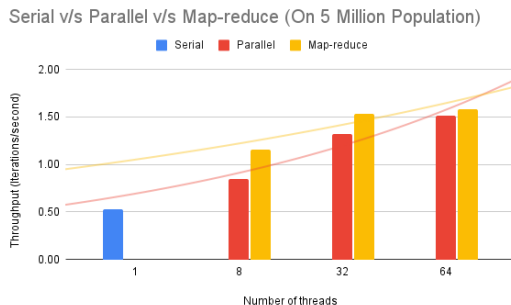


Figure 3: Serial v/s Parallel v/s Map-reduce (for 5 million population).

The Rust language promotes what is called ‘*fearless concurrency*’ by virtue of its distinctive memory management approach which guarantees memory safety especially for concurrent accesses (Klabnik and Nichols, 2019). The Rust ecosystem provides many libraries (*crates* as they are called) to harness parallelism. To represent the grid, we are using concurrent hashmap implementation supported by *dashmap* crate (Dashmap, 2023). Another such crate is *Rayon* (Rayon, 2022) which provides a data-parallelism library for data-race free computations. Using these libraries, we evaluated two different ways for harnessing multi-core computers namely *parallel iteration*, and *map-reduce implementation*.

4.1 Parallel Iteration

Rayon has a module called as *par_iter* to help us achieve the results. The *par_iter* module spawns multiple threads and then executes different data segments on different CPU cores using these threads for getting higher throughput. Rayon makes it easy converting a sequential iterative computation into parallel iterative one, just by substituting *iter* module with *par_iter* module.

4.2 Map-Reduce Implementation

Map-reduce is a dual-operation data-parallel technique used for processing large collection data. The *map* operation assigns data elements to available compute elements in batches and waits till all data elements are processed. The *reduce* operation collects these results and summarizes (or reduces) them into a desired form. Rayon provides map-reduce operation on collections (Dean and Ghemawat, 2008). EpiRust employs a *map* function on each agent such that the agent goes through evaluation and state update. The *reduce* operation collects older and newer locations of each agent along with few other details.

The next task was to evaluate throughput of these two implementations. The Figure 3 shows results of running parallel and map-reduce based simulations for a hypothetical society of 5 million agent population. Based on these results, the map-reduce implementation performs slightly better than the parallel implementation, especially for the lower number of cores. With higher cores, the throughput becomes plateaued. This observation became a challenge for our next goal to run simulations for large cities like Mumbai (with 12+ million population), and suggested looking for approaches like a distributed implementation. A stable, performant, and flexible distributed implementation could scale from tens of mil-

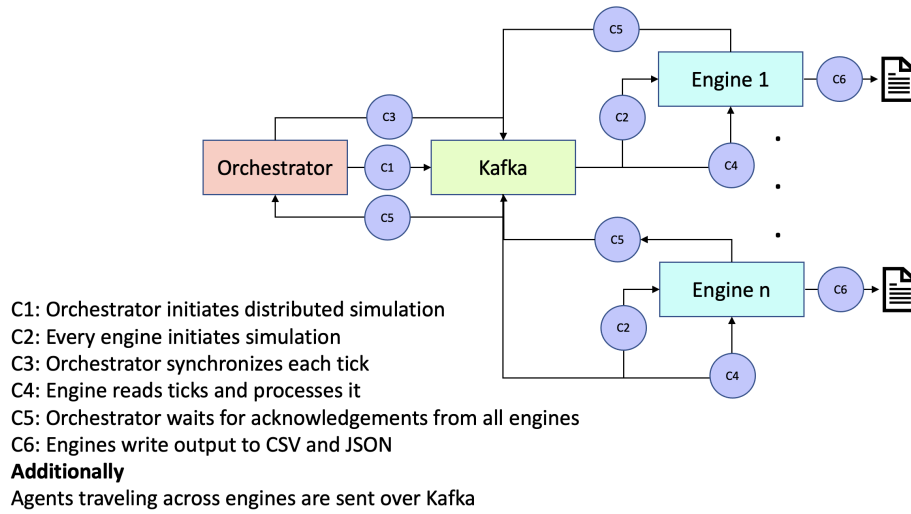


Figure 4: System Architecture.

lions to hundreds of millions.

5 DISTRIBUTED IMPLEMENTATION

Simulating large cities as a serial and single process has been a challenge due to steeper memory and processor requirements. On the CPU front, combining insights from Figures 2 and 3 tells that the throughput would drop substantially for large cities like Mumbai. In addition, the throughput per core is far better for smaller agent populations. In addition, EpiRust consumes more memory for the traditional double-buffering approach.

These observations prompted us to organize a large virtual society into smaller hierarchical units, and then running simulations for these smaller units becomes an attractive possibility for scaling, in particular scaling out.

In India, cities have smaller administrative units called ‘wards’. Wards have diverse population density, public transport facilities, initial number of infections, and so on. EpiRust configuration accepts these parameters. Thus, a monolithic EpiRust process instance is broken into multiple instances called *EpiRust Engines* as depicted in Figure 4. Usually, one engine is assigned per administrative unit. Each of these engines can now use multiple CPU cores in parallel.

Once generalized, the approach could be applied upwardly from cities to states to an entire country. For such larger experiments, the computing infrastructure also needs to scale beyond a single powerful computer to a cluster of computers, which suggests the need for

a distributed implementation of EpiRust.

5.1 Traveling and Commuting Across Engine Instances

In the previous EpiRust model, a city was a monolithic representation such that agents would live entirely in that city. Agents would commute between home and work as per their schedules. With multiple engine instances, each engine instance would have a separate grid with its own residential, transport, work areas and hospitals. In addition, for large cities like Mumbai, an agent could live in an engine representing the ward, and could commute to another ward. Hence modeling of commute in EpiRust is an important part of simulating urban scenarios.

5.2 Using Apache Kafka as a Distributed Event Store

To implement commuting, an engine needs to communicate and coordinate with other engines (please see C2 and C4 in Figure 4). We see this communication as an event publish-subscribe based approach, for which we use Apache Kafka, an open source, distributed event store and stream processing platform. An agent or set of agents leaving the engine can be seen as an event or message over a Kafka instance. For load distribution, we have assigned a topic per engine which that engine listens to. Each engine sends messages describing the leaving agents to the topics of destination engines. Engines receiving the messages would spawn agents with specified details.

During the COVID-19 epidemic, many Indian cities witnessed unfortunate migrant workers en

masse going back to their hometowns. EpiRust can simulate such travel and/or migration with appropriate configuration.

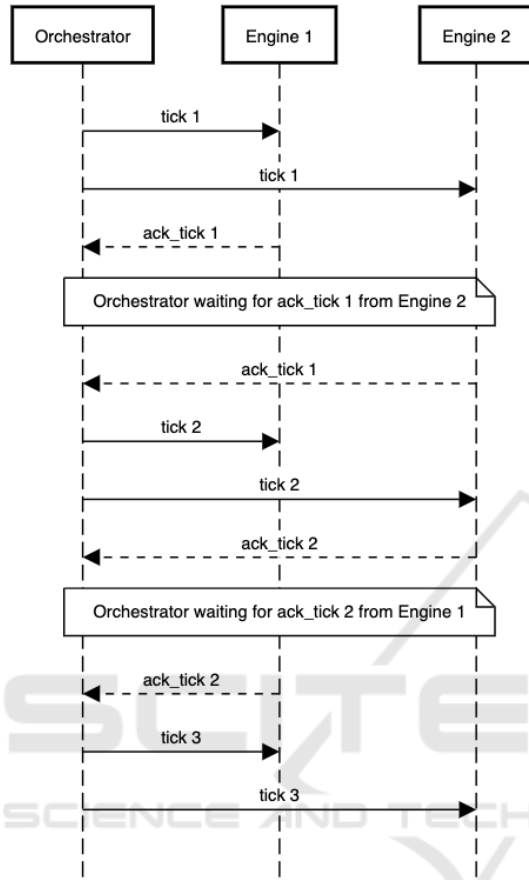


Figure 5: Barriers to synchronize commuter population across engines in a 24-hours (or ticks) day.

5.3 Orchestration

In a distributed setup, engines could run at differential throughput depending upon their population sizes as seen in Figure 3. However, if the engines are required to support commute, travel, and migration, then the engines need to synchronize at each time step or tick for transferring agents amongst themselves. For example, if an agent were to leave a source engine at time t , and to reach its destination engine at time $t+k$, assuming the travel takes k units of time, then the destination engine needs to be synchronized at $t+k$ time. However, if the destination engine has fewer agents and hence higher throughput, then it must wait for other engines to join it at the tick $t+k$. For ensuring this barrier, EpiRust has a component namely Orchestrator, and it is commonly required in a distributed simulation setup as discussed in a survey pa-

per on co-simulation (Gomes et al., 2018).

The orchestrator is a process doing multiple jobs. First, it sends the initialization configuration to all engines to instantiate their geographies and disease dynamics (please refer to C1 in Figure 4). Second, it synchronizes all the engines twice in 24 ticks, when agents travel to-and-fro engines using kafka. It sends a vector tick over Kafka (please refer to C3 in Figure 4), and then on receiving this tick, each engine starts its execution. During this tick, each engine executes routine for all its agents, sends an acknowledgement after completion, and waits for the next tick from the orchestrator (please refer to C4 in Figure 4). The orchestrator waits for acknowledgements, and once received from all engines, only then it sends the next tick (please refer to C5 in Figure 4).

Here, the slowest engine determines the overall throughput of the distributed setup. Hence, it is important to balance throughput of all engines by adjusting their parameters and by assigning appropriate compute resources. In this way, each engine could finish a time step or tick roughly in the same duration, thereby the time spent in synchronization or barrier at the end of that tick could be minimized.

In our earlier distributed EpiRust implementation, we faced multiple challenges to simulate populations beyond 10 million. Here, we discuss two critical challenges related to communication and coordination across engines as shown in Figure 5.

The first challenge was about varying engine throughput caused by different agent populations assigned to them. This variation made faster engines with lower populations to wait for slower engines with higher populations, at the commute barrier 5.

The second challenge was related to the synchronization bottleneck while using kafka. The root cause of this problem was that all engines were simultaneously publishing and receiving all the messages over a shared kafka topic. Each engine had to scan through all messages to identify those meant for it, and this approach added overheads and thereby wait-times for the engines. These wait-times not only degraded the performance of engines but also introduced reliability issues due to intermittent broker connection failures.

We fixed these failures by increasing the polling interval and session timeout duration for consumers. To improve the performance, scaling of Kafka became necessary. As every engine is in a separate consumer group, we could not increase the number of consumers (or partitions). So, we created separate topics, one for every engine to consume the commuters' information. With this change, an engine now receives the relevant messages meant only for itself instead of all the messages meant for all the en-

gines. Thereafter, to improve load balancing, we distributed these topics across multiple brokers. This approach helped in reducing the message consumption overhead, thereby yielding approximately 50% performance improvement for large populations as discussed in the results section 6.2.

5.4 Cloud-Ready Deployment Setup

Depending upon the size of simulation, deploying and managing a large-scale distributed EpiRust on a cluster of workstations or a cloud service could be a non-trivial task. The recent advancements in virtualization technologies such as *containers* and *Kubernetes* take away much of the complexity on the user-side for efficient use and management of computing resources.

Containers are execution environments, used for packaging and running applications in isolation. A container wraps all the application dependencies such as application binaries, configuration files, third party libraries, and operating system required for its execution. Being an isolated environment, a containerized application is abstracted from the underneath infrastructure and operating system making containers portable across platforms. With these features, containers aid in handling fluctuations in workload and help in scaling the application. These containers need to be managed for scaling and resilience and that is where Kubernetes comes into picture.

Kubernetes is ‘an open-source system for managing containerized applications across multiple hosts’ (kubernetes.io, 2022). It provides basic mechanisms for deployment, maintenance, and scaling of applications. It helps in running distributed systems resiliently. EpiRust can be deployed on any infrastructure with a Kubernetes service which includes all major cloud providers as well as any locally set up Kubernetes clusters. Each EpiRust engine instance can be configured to represent a different geographical region. These instances generate output files during the simulation runs. For storage of these configuration and output files, *Volumes* are used. Volumes allow us to store files and share them across containers. To install the engine instances and the orchestrator, we use *Helm*. The application is packaged using *Helm Chart* to start/stop simulation with a single command, and provide a way of pre-processing, post-processing, and cleanup of data via hooks. Logging and monitoring are essential aspects for observability of the deployed software. We use other open-source software like Prometheus, Grafana, ELK stack for logging and monitoring purposes for resource optimization, automated alerts and debugging. With containers and Kubernetes support, efforts for a large-scale

EpiRust deployment are reduced. This substantially eases scheduling of simulation experiments.

In the following section, we discuss a few large-scale experiments and their results.

6 EXPERIMENTS AND RESULTS

In this section, we describe how the experiments can now be run in parallel and distributed modes for large cities, and that too at higher throughput. We begin by sharing the compute infrastructure used for these experiments. It is followed by the details of baseline and intervention scenarios for simulation experiments for Pune and Mumbai cities. Finally, we discuss the results of these experiments and compare them with the results from earlier EpiRust paper.

6.1 Experiment Infrastructure

As discussed in an earlier section 5.4, the containers and Kubernetes pods have made EpiRust a cloud-ready application. Our experiments were scheduled on an in-house or local computing infrastructure and on a public cloud infrastructure.

Local Infrastructure: The in-house infrastructure was configured as a local Kubernetes cluster. The cluster had 5 compute nodes, each with an *AMD Ryzen 2700X* microprocessor providing 8-cores and 16-threads, and 64 gigabytes of memory. The experiments using parallel and map-reduce versions of EpiRust could run on multiple CPU cores. In a distributed setup managed by Kubernetes, each engine was run in a Kubernetes pod and got a CPU core as per the job configuration. An upper bound for memory consumption could be specified, and in its absence, an engine could consume up to the entire available memory. Three brokers each for Apache Kafka and Apache Zookeeper were run in the same cluster with their own Kubernetes pods.

Cloud Infrastructure: We had set up a fifteen-node Kubernetes cluster on *Elastic Kubernetes Service* (EKS) of Amazon Cloud Services (AWS). These nodes were based on the AWS’ *c5a.4xlarge* instances, each equipped with 16 *vCPUs* and 32 gigabytes of memory. The reason for choosing this instance type was that EpiRust is compute-intensive and not memory or I/O intensive. *AWS Elastic File System* (EFS) was used for storage. Three brokers each for Apache Kafka and *Apache Zookeeper* were deployed in the same cluster with their own Kubernetes pods.

All the simulations for Mumbai and Pune in the *parallel mode* were run with eight CPU cores.

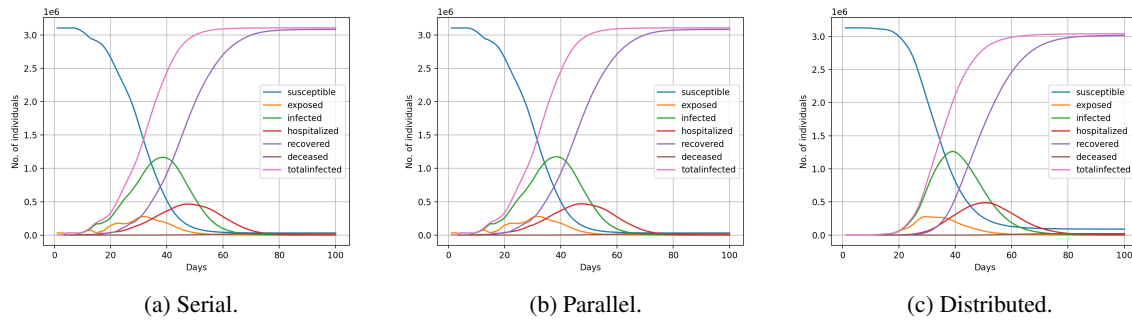


Figure 6: Epidemic Curves of Serial, Parallel, and Distributed Stochastic Simulations of Pune City.

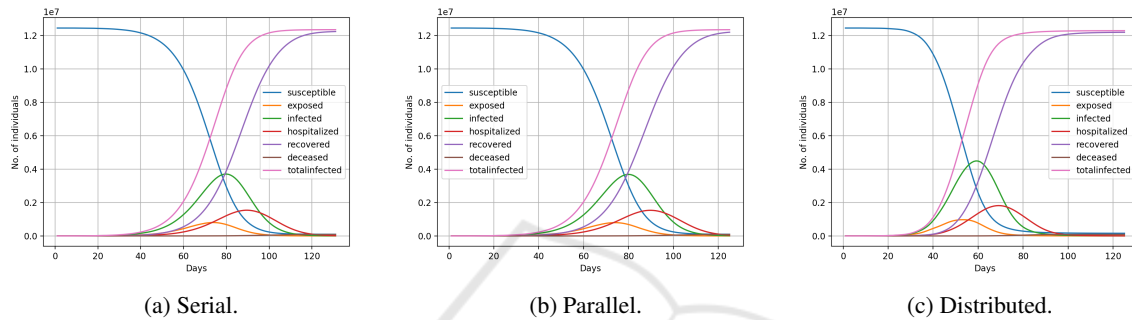


Figure 7: Epidemic Curves of Serial, Parallel, and Distributed Simulations of Mumbai City.

6.2 Experiments

Here, we discuss the experiments for simulating an unchecked COVID-19 infection spreading through two large cities, Pune and Mumbai. In addition, we have an experiment for a hypothetical large society of 100 million population, mainly to test scalability of EpiRust.

6.2.1 Pune

To run simulations for Pune city in distributed mode, we created 15 engine instances, one for each ward (administrative unit) in the city. These wards have populations varying within a smaller range between 146,333 and 253,778 with mean at 208,809. We have modeled commutation within and across wards. Within a ward, approximately 80% of the working agents use public transport and the rest use private transport. Across wards, all agents use public transport. The model assumes that 100 agents from every ward commute to every other ward, making it 1400 agents per ward and 21000 total across the city.

6.2.2 Mumbai

Mumbai has 24 administrative wards according to the government health department (Mumbai, 2011). The population across wards ranges between 127,290 and

941,366 with an average of 518,432. The model assumes agent commuting patterns like Pune.

6.2.3 A Hypothetical Society of 100-million Population

This section describes our experiments to stress-test the setup for a hypothetical society of 100 million population. To test the scalability limits of distributed EpiRust, we ran the simulation with 100 million population spread over 100 EpiRust engines each with a population of 1 million agents on the AWS infrastructure. We spawned 15 nodes with c5a.4xlarge with 100 engine instances. Each engine was equipped with 2 CPU cores and 2 GB of maximum memory.

6.3 Results

The results of the serial, parallel, and distributed setup are discussed below. For validation, we compared the Mumbai and Pune results with respective serial runs. The shapes of epidemic curves (epi-curves) are as seen in Figures 6 and 7. However, the distributed simulation of Pune shows an altered shape and left shifted exposed curve in Figure 6c than in Figures 6a and 6b. Like Pune, we can observe a leftward shift in all curves, and taller and steeper infected peak in Figures 7c. Our explanation is that the agents form

denser and localized contact networks in a distributed setup, which alters the nature of disease contagion.

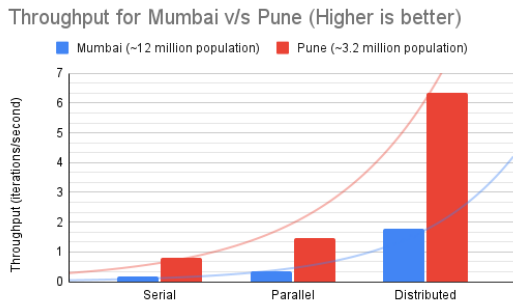


Figure 8: Comparison of Throughput of Serial, Parallel, Distributed Simulations of Mumbai and Pune Cities on AWS Cloud Infrastructure.

To understand the throughput of this implementation, Figure 8 compares the throughput numbers of simulations for Mumbai and Pune cities in serial, parallel, and distributed modes. The Mumbai simulation in the parallel mode is approximately three times faster than in the serial mode, and more than ten times faster in distributed mode. Improvements for Pune simulations are in the similar order, which is two times and eight times faster in parallel and distributed modes respectively than the serial mode.

Earlier for a 100 million population, a 45-day simulation (1080 ticks) had finished within 10920 seconds (approximately 3 hours), with throughput of 0.09 iterations per second. This implementation however had suffered a communication bottleneck discussed in section 5.3. After fixing this bottleneck, the same simulation finished in 3960 seconds (approximately 1 hour 6 minutes) with throughput close to 0.27 iterations per second, which is roughly 2.7 times higher than the earlier implementation.

7 CONCLUSION AND FUTURE WORK

With parallel and distributed implementations of EpiRust, we could transition to a faster policy evaluation regime for large cities like Mumbai and Pune. The speedup from the serial to distributed has improved between 8x and 10x. This transition should help us in moving towards even larger experiments along the geographical hierarchy of cities, districts/counties, states, and the country. In our plans, we intend to identify and fix computation and communication bottlenecks of the current version. We hope that widening these bottlenecks could help

EpiRust scale to the population of a billion agents. Finally, we would like to understand the stability of these experiments using statistical methods.

8 REPRODUCIBILITY

In order to repeat the experiments described in this paper, one can refer to the source code of EpiRust which is available at <https://github.com/thoughtworks/epirust/tree/icaart2023>. Details about the experiments, their setup and configurations can be found here: <https://github.com/thoughtworks/epirust/tree/icaart2023/experiments>.

ACKNOWLEDGEMENTS

The authors would like to thank Dhananjay Khaparkhantkar for his contributions in development, and Shivir Chordia (Azure Business Lead of Microsoft India) for his support of cloud compute resources during the prototyping phase of EpiRust.

REFERENCES

- Abar, S., Theodoropoulos, G. K., Lemariner, P., and O'Hare, G. M. (2017). Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review*, 24:13–33.
- Antelmi, A., Cordasco, G., D'Auria, M., Vinco, D. D., Negro, A., and Spagnuolo, C. (2019). On evaluating rust as a programming language for the future of massive agent-based simulations. In *Asian Simulation Conference*, pages 15–28. Springer.
- Bisset, K. R., Chen, J., Feng, X., Kumar, V. A., and Marathe, M. V. (2009). Epifast: a fast algorithm for large scale realistic epidemic simulations on distributed memory systems. In *Proceedings of the 23rd international conference on Supercomputing*, pages 430–439.
- Brauer, F. (2008). Compartmental models in epidemiology. In *Mathematical epidemiology*, pages 19–79. Springer.
- Chapuis, K., Taillandier, P., and Drogoul, A. (2022). Generation of synthetic populations in social simulations: A review of methods and practices. *Journal of Artificial Societies and Social Simulation*, 25(2):6.
- Childs, M. L., Kain, M., Kirk, D., Harris, M., Couper, L., Nova, N., Delwel, I., Ritchie, J., and Mordecai, E. (2020). The impact of long-term non-pharmaceutical interventions on covid-19 epidemic dynamics and control. *medRxiv*.
- Collier, N. (2003). Repast: An extensible framework for agent simulation. *The University of Chicago's Social Science Research*, 36:2003.

- Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., and Spagnuolo, C. (2013). Bringing together efficiency and effectiveness in distributed simulations: the experience with d-mason. *Simulation*, 89(10):1236–1253.
- Cosenza, B., Popov, N., Juurlink, B., Richmond, P., Chimeh, M. K., Spagnuolo, C., Cordasco, G., and Scarano, V. (2018a). Openabl: a domain-specific language for parallel and distributed agent-based simulations. In *European Conference on Parallel Processing*, pages 505–518. Springer.
- Cosenza, B., Popov, N., Juurlink, B. H. H., Richmond, P., Chimeh, M. K., Spagnuolo, C., Cordasco, G., and Scarano, V. (2018b). Openabl: A domain-specific language for parallel and distributed agent-based simulations. In *Euro-Par*.
- Dashmap (2023). Dashmap: Blazingly fast concurrent map in rust.
- de Aledo Marugán, P. G., Vladimirov, A., Manca, M., Baugh, J., Asai, R., Kaiser, M., and Bauer, R. (2018). An optimization approach for agent-based computational models of biological development. *Adv. Eng. Softw.*, 121:262–275.
- Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.
- Dias, S., Sutton, A. J., Welton, N. J., and Ades, A. E. (2013). Evidence Synthesis for Decision Making 3: Heterogeneity—Subgroups, Meta-Regression, Bias, and Bias-Adjustment. *Medical Decision Making*, 33(5):618–640.
- Eubank, S., Guclu, H., Anil Kumar, V., Marathe, M. V., Srinivasan, A., Toroczkai, Z., and Wang, N. (2004). Modelling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180–184.
- Gilbert, N. and Terna, P. (2000). How to build and use agent-based models in social science. *Mind & Society*, 1(1):57–72.
- Gomes, C., Thule, C., Broman, D., Larsen, P. G., and Vangheluwe, H. (2018). Co-simulation: a survey. *ACM Computing Surveys (CSUR)*, 51(3):1–33.
- Gulyás, L. (2005). *Understanding Emergent Social Phenomena*. PhD thesis, Computer and Automation Research Institute, Budapest.
- Hash (2022). hash.ai. <https://hash.ai/>.
- Hessary, Y. K. and Hadzikadic, M. (2017). Role of Behavioral Heterogeneity in Aggregate Financial Market Behavior: An Agent-Based Approach. *Procedia Computer Science*, 108:978–987.
- Jaffry, S. W. and Treur, J. (2008). Agent-based and population-based simulation: A comparative case study for epidemics. In *Proceedings of the 22nd European Conference on Modelling and Simulation*, pages 123–130. Citeseer.
- Kagho, G. O., Meli, J., Walser, D., and Balac, M. (2022). Effects of population sampling on agent-based transport simulation of on-demand services. *Procedia Computer Science*, 201:305–312.
- Kermack, W. O. and McKendrick, A. G. (1927). A contribution to the mathematical theory of epidemics. *Proceedings of the royal society of london. Series A, Containing papers of a mathematical and physical character*, 115(772):700–721.
- Kerr, C. C., Stuart, R. M., Mistry, D., Abeysuriya, R. G., Rosenfeld, K., Hart, G. R., Núñez, R. C., Cohen, J. A., Selvaraj, P., Hagedorn, B., George, L., Jastrzębski, M., Izzo, A. S., Fowler, G., Palmer, A., Delport, D., Scott, N., Kelly, S. L., Bennette, C. S., Wagner, B. G., Chang, S. T., Oron, A. P., Wenger, E. A., Panovska-Griffiths, J., Famulare, M., and Klein, D. J. (2021). Covasim: An agent-based model of covid-19 dynamics and interventions. *PLOS Computational Biology*, 17(7):1–32.
- Klabnik, S. and Nichols, C. (2019). *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.
- Klabunde, A. and Willekens, F. (2016). Decision-Making in Agent-Based Models of Migration: State of the Art and Challenges. *European Journal of Population*, 32(1):73–97.
- Kshirsagar, J. K., Dewan, A., and Hayatnagarkar, H. G. (2021). EpiRust: Towards a framework for large-scale agent-based epidemiological simulations using rust language. In *Linköping Electronic Conference Proceedings*. Linköping University Electronic Press.
- kubernetes.io (2022). Kubernetes (k8s).
- Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527.
- Matsakis, N. D. and Klock, F. S. (2014). The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104.
- Mumbai, M. C. O. G. (2011). Mumbai population breakup by administrative wards.
- Parker, J. and Epstein, J. M. (2011). A Distributed Platform for Global-Scale Agent-Based Models of Disease Transmission. *ACM Transactions on Modeling and Computer Simulation*, 22(1):1–25.
- Parry, H. R. and Bithell, M. (2012). Large scale agent-based modelling: A review and guidelines for model scaling. *Agent-based models of geographical systems*, pages 271–308.
- Patlolla, P., Gunupudi, V., Mikler, A. R., and Jacob, R. T. (2004). Agent-based simulation tools in computational epidemiology. In *International workshop on innovative internet community systems*, pages 212–223. Springer.
- Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017). Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 256–267, Vancouver BC Canada. ACM.
- Rayon (2022). Rayon: Simple work-stealing parallelism for rust.
- Snehal Shekatkar, Bhalchandra Pujari, Mihir Arjunwadkar, Dhiraj Kumar Hazra, Pinaki Chaudhuri, Sitabhra Sinha, Gautam I Menon, Anupama Sharma, and Vishwesh Guttal (2020). Indsci-sim a state-level epidemiological model for india. Ongoing Study at <https://indscicov.in/indscisim>.

- Taillandier, P., Vo, D.-A., Amouroux, E., and Drogoul, A. (2010). Gama: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 242–258. Springer.
- Tisue, S. and Wilensky, U. (2004). Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA.
- White, S. H., Del Rey, A. M., and Sánchez, G. R. (2007). Modeling epidemics using cellular automata. *Applied mathematics and computation*, 186(1):193–202.
- Wildt, T. (2015). Heterogeneity, agent-based modelling and system dynamics - A study about the effects of including adopter heterogeneity in diffusion of innovations models and the consequences on paradigm choice. *Unpublished*. Publisher: Unpublished.
- Xiao, J., Andelfinger, P., Cai, W., Richmond, P., Knoll, A., and Eckhoff, D. (2020). Openablext: An automatic code generation framework for agent-based simulations on cpu-gpu-fpga heterogeneous platforms. *Concurrency and Computation: Practice and Experience*, 32(21):e5807.

