Generalizing Bulk-Synchronous Parallel Processing for Data Science: From Data to Threads and Agent-Based Simulations

ZILU TIAN, EPFL, Switzerland
PETER LINDNER, EPFL, Switzerland
MARKUS NISSL*, TU Wien, Austria
CHRISTOPH KOCH, EPFL, Switzerland
VAL TANNEN, University of Pennsylvania, USA

We generalize the bulk-synchronous parallel (BSP) processing model to make it better support agent-based simulations. Such simulations frequently exhibit hierarchical structure in their communication patterns which can be exploited to improve performance. We allow for the creation of temporary artificial network partitions during which agents synchronize only locally within their group in a way that does not compromise the correctness of a simulation. We have built a distributed engine, CloudCity, which uses this idea to improve the locality of computation, communication, and synchronization in such simulations. We experimentally evaluate the performance of our system on a benchmark of simulation workloads and compare it against other popular BSP-like systems, obtaining insights into the impact of various system design choices and optimization on simulation engine performance.

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **Information systems** → *Information retrieval*; *Query languages*; • **Computing methodologies → Simulation support systems**.

Additional Key Words and Phrases: agent-based simulations, distributed systems, bulk-synchronous parallel processing, compilation, query languages

1 INTRODUCTION

Agent-based simulations are simulations in which a number of agents, each with their own thread, code, and state, interact in a virtual environment. They play an increasingly important role in the social sciences [6, 24, 26, 27, 36, 41, 42, 44, 63, 69, 70], sustainability research [14, 20, 46, 47, 50, 52, 59, 60], economics and finance [10, 15, 25, 29, 58, 71, 73], epidemiology [1, 40, 43, 49, 64, 66], population dynamics [11, 21, 38], and entertainment (gaming [5]). Agent-based simulations have become important tools to help make decisions of monumental importance to modern society.

The popularity of the agent-based simulation paradigm is mainly due to its great expressive power. An agent may execute arbitrary code, affecting its state as well as the state of the virtual world. Developing precise simulations is relatively simple: The desired behavior can be coded directly, rather than requiring the crafting of complex mathematical models (such as systems of partial differential equations) that describe the scenario indirectly or in the aggregate.

Agent-based simulations are increasingly prevalent as building blocks of data science pipelines and impose significant data science and data engineering challenges. They are data-intensive, as a simulation's state can be very large. Simulations can be thought of as operators (even *queries*) to be composed with other data management and analysis operations in the pipeline. This creates a need for suitable optimization techniques. A simulation's start state and parameterization come from databases and analytics upstream. The analysis of simulation traces and outcomes downstream requires complex data transformation, aggregation, time series analysis, and machine learning. As such, agent-based simulations are of interest to the data management research community.

*Work done while the author was at EPFL.

Authors' addresses: Zilu Tian, EPFL, Lausanne, Switzerland, zilu.tian@epfl.ch; Peter Lindner, EPFL, Lausanne, Switzerland, peter.lindner@epfl.ch; Markus Nissl, TU Wien, Vienna, Austria, markus.nissl@tuwien.ac.at; Christoph Koch, EPFL, Lausanne, Switzerland, christoph.koch@epfl.ch; Val Tannen, University of Pennsylvania, Philadelphia, USA, val@cis.upenn.edu.

Example 1.1. A COVID study by epidemiologists at Imperial College made public in the spring of 2020 [31, 40] garnered worldwide media attention since it predicted that the then-prevalent strategy of the British and US governments to approach COVID by inaction, hoping for the population to achieve herd immunity, could lead to more than half a million deaths in a relatively short time span in the UK alone, and to about 2.2 million deaths in the US. These results were obtained by adapting an agent-based simulation model first developed for avian influenza (H5N1) in Southeast Asia [30]. This simulation model used rich information about social networks, commute patterns, occupations, social classes, and the school system on the level of individuals. In many cases, data was unavailable or incomplete – for instance, population density data as a basis of social network data in rural areas was reconstructed from satellite images – and gaps in datasets were filled using a combination of clever data engineering and auxiliary simulations. The document [30] describes this highly complex data science pipeline that combines a number of simulation models.

Challenges. While a number of agent-based simulation toolkits have been made available [53], most seem to be the outgrowth of simulation projects by domain scientists, and to this day, agent-based simulations have seen very little foundational research from core computer science. This leads us to two *main challenges*:

(1) We need a clean abstraction of agent-based simulations, with a precise semantics that helps us understand the commonalities with and differences from established models of distributed computation. This allows us to draw from previous research and facilitates further work.
(2) We need to create and understand foundations and architectures for systems (engines) for running agent-based simulations at *large scale* – for instance, to simulate billions of humans in an economic or epidemics simulation.

Scaling such simulations up and out is challenging. While simulations are naturally massively parallel (they are programmed to run as large collections of parallel threads), what makes the problem difficult is the flexibility of the agent-based simulation paradigm. The amount of necessary communication, coordination, and synchronization among agents and their environment is frequently very high when compared to the amount of local computation performed by agents. Also, given the different roles played by the agents in a simulation, the computation and communication patterns among the set of agents in a simulation may be highly heterogeneous and skewed. Thus, the key challenge is to tame the complexity of synchronization between threads (agents) while ensuring consistency according to a suitable semantics. A main goal is to transform the agent-based simulation problem into one that can profit from optimized synchronization, “embarrassing” parallelism, increased data locality, and, ultimately, if possible, well-established, optimized data-parallel processing architectures and systems.

Traditionally, executing large numbers of heterogeneous threads belongs to the domain of operating systems. However, general-purpose operating systems (1) do not satisfactorily scale to today’s needs of agent-based simulations (often billions of agents / threads), and (2) do not take advantage of the communication and synchronization patterns specific to agent-based simulation. Allowing and requiring agents to perform unconstrained interprocess communication and to implement their own synchronization protocols leaves much performance potential untapped.

One important simplifying factor is that, in many domains, the builders of simulations desire semantics of a highly synchronous “round-based” flavor (like in many board games), which is relatively intuitive and easy to understand. In a round, agents act individually: they perform computations, process incoming messages, and send messages of their own.[1] This is followed by global synchronization and message reshuffling and delivery across the entire simulation at the end of each round. The amount of computation done by each agent in each round is typically moderate, making such simulations relatively synchronization-heavy.

Bulk-synchronous parallel processing [67] (BSP) systems such as Map-Reduce [23], Spark [72], and Pregel [45] have become an industry standard for processing massive amounts of data in parallel, and are successfully used in many big data processing and data science applications. The BSP model seems well-aligned with the round-based simulation model just described. This must be weighed against the above-observed heavy skew present in many simulations: Vanilla BSP is considered not to be as well-suited if the amount of work to be done to the data segments is highly skewed. Also, given that the skew in such simulations may be very volatile and the simulation semantics prescribes very frequent global synchronization (thus the embarrassingly parallel phases between synchronizations are relatively short), one cannot expect wonders from established load-balancing techniques. Still, BSP, as a family of architectures, remains a prime candidate for designing agent-based simulation engines.

Large simulations frequently exhibit (temporary) hierarchies of groups of agents such that the communication latency constraints are hierarchical – the delivery latencies for messages across group boundaries are allowed to be greater than for message delivery among agents in the same group. For instance, in an epidemics simulation, agents may be grouped by city or country, with infections spreading between groups more slowly than within a group, due to the latencies caused by travel or quarantine measures. Even if the simulation semantics is based on global synchronization between rounds, to simulate actions of varying duration, agents may be inactive for some rounds or send messages that take several rounds to arrive at their destination, to approximate varying amounts of passed real-time. This means that the need for frequent global synchronization in an actual simulation may be less stringent than its abstract semantics dictates. A simulation engine that can leverage this fact may be able to partition its workload and perform the necessary synchronizations more efficiently: It may execute rounds and synchronize the agents within a group faster as data is more local and synchrony (consensus) needs to be achieved for smaller groups of agents than otherwise, while performing global synchronization less frequently and preserving the desired semantics.

To be able to leverage these implicit hierarchies in simulations for obtaining performance benefits, we need to make research progress on a number of fronts, including ($*$) a suitable, easy-to-use programming model that allows expressing waiting through rounds and relaxed message delivery latencies in a way that can be understood and leveraged by the system, ($*$) a formulation of a simulation semantics that allows showing that certain relaxed synchronization choices are correct and equivalent to full round-based synchronization, ($*$) algorithms for creating hierarchies of agents that allow for the semantically correct weakening of global synchrony, and techniques for exploiting the performance benefits that become thus obtainable, and ($*$) a system architecture that supports the optimization and execution of simulations while embedded within a data science pipeline. Progress on this front may allow for faster, larger, and thus more accurate agent-based simulations, providing better insights and predictions.

Contributions. We propose a simulation semantics in which we re-purpose the abstraction of partition tolerance in distributed systems. Network partitions are usually outside of our control, and a distributed system needs to cope with them. In this work, we turn them into a tool for more

[1]For simplicity, but without loss of generality, we assume a shared-nothing model in which each agent has its private state, and agents interact only by messaging; the purpose of synchronization is purely to ensure that message orderings (with respect to round boundaries) are consistent with the simulation semantics.

efficient simulations. We seek to form groups of agents such that synchronized communication across group boundaries is not required in every round. The system is tolerant to a network partition along group boundaries for a certain number of rounds, as messages sent across the boundary during the partition do not need to be delivered immediately. So, conceptually, we can create a brief artificial network partition during which we do not need to synchronize globally (just locally, among the agents of each group) without compromising the correctness of the simulation. To this end, we formalize the notion of K-availability, which models the acceptable latency by which pairs of agents may communicate. Distinguishing between agents within a group that synchronize at a high rate and those synchronizing across group boundaries at a lower rate permits optimizing the execution of a large simulation. This is achieved by aligning the hierarchy of agent groups with the hardware and memory hierarchy and exploiting (data) locality. The simulation is equivalent to an unaltered run in which no partition is introduced. We formalize and develop this idea into a model of generalized BSP computation that is suitable for efficiently processing agent-based simulations.

Based on this model, we develop a system architecture for agent-based simulation engines, which we have also implemented in a scalable engine called CloudCity.[2] The optimizations supported by this system include: (∗) Groups of collocated agents are de-parallelized by turning threads into coroutines via thread merging. The degree of parallelism is reduced to better match the number of available hardware threads. A hierarchy of agents can be automatically aligned to the hardware hierarchy. (∗) The round-based semantics dictates that messages sent in one round arrive at the beginning of a later round. This does not immediately allow to turn local message passing into direct read and write accesses to other agents' states without violating event orders under our semantics. We introduce an efficient double-buffering mechanism that, however, allows to turn message passing into direct memory accesses, and thus remove significant compute and communication overheads. In our architecture, a simulation is presented as a pipeline operator that maps an initial simulation state (a set of initial agent states) to a time series of simulation states that captures the simulation run. This time series can be easily queried by a collection-based query language, and our system supports the optimization of simulation execution and querying by deforestation; that is, the recording (logging) of the time series as the simulation is running is optimized to only include the output required downstream.

We compare CloudCity with current state-of-the-art distributed systems that support BSP-like computations, specifically Spark [72], GraphX [37], Flink Gelly [12, 33], and Giraph [3, 45]. We use representative workloads selected from fields where agent-based simulations are prevalent to benchmark the relative performance of these systems. Since such systems have different programming models, we explain important details concerning how we implemented the workloads to ensure that the agent code in a workload behaves the same across systems. Regarding the relative performance of these systems, even though GraphX, Flink Gelly, and Giraph are all designed for large-scale graph processing, our intuition is that their performances when executing agent-based simulations will depend on the underlying system designs. To convey this intuition, we summarize various system features that are most relevant to obtaining efficient executions of agent-based simulations, including in-place updates and efficient local messaging, and investigate the applicability of these features for each system. This high-level analysis is accompanied by empirical evaluation, which demonstrates that the relative performance of different systems can indeed differ by orders of magnitude due to their alternative design choices.

For the experiments, we start with tuning each system, for example, by varying the number of workers per machine and adopting their best configuration for other experiments. As the number of agents and machines increases, our experimental results show that across all workloads, CloudCity

[2]The Cloud City in Star Wars is located "hierarchically" above BeSPin.

is 2× faster than Flink Gelly, one to two orders of magnitude faster than Spark and GraphX, and more or less on par with Giraph. After that, we run microbenchmarks to stress test each system, by increasing the number of messages sent per round to other agents as well as the number of rounds until the computation is resumed. As expected, the execution time of all systems increases as the number of messages increases to 30×; CloudCity and Giraph have similar performance, and both systems are 2-10× faster than Flink Gelly, and over 100× faster than Spark and GraphX. When introducing idle periods of up to 20 rounds, the overall average time per round drops for all systems, but to different extents. CloudCity has built-in support to speed up intermittent computation, which leads to a performance advantage. With idle periods of 20 rounds, CloudCity is over an order of magnitude faster than both Giraph and Flink Gelly, and two orders of magnitude faster than Spark and GraphX. Finally, we evaluate CloudCity-specific optimizations and exploit hierarchical communication latency constraints with K-availability separately. Direct memory access is up to 2× faster than messaging. Deforestation can potentially eliminate almost all of the overhead of materializing the time series for a simulation, and the speedup depends on the selectivity of the user query and the number of user-defined attributes. In our experiments, deforestation improves performance by up to 5×. For hierarchical communication, synchronizing different components only every 20 rounds also improves performance by 2× in all workloads.

The rest of the paper is organized as follows. In Section 2, we introduce our programming model, which showcases how users can interact with our distributed simulation engine through a self-contained example. In Section 3, we delve into the computational model of our system, explaining how computation proceeds in a distributed simulation and how agents synchronize. Our system architecture is designed for efficient execution of our computational model, as highlighted in Section 4. In Sections 5 and 6, we provide details on translating our DSL to Scala and exploring system optimizations, respectively. We then describe our experimental workloads and compare our system with current state-of-the-art BSP systems, analyzing both high-level insights and empirical results in Section 7. In Section 8, we discuss existing distributed agent-based simulation frameworks and their scalability bottlenecks. Finally, we summarize key insights and future work in Section 9.

2 PROGRAMMING MODEL

At its core, CloudCity is a distributed agent-based simulation engine. Users define parallel agent programs using a domain-specific language (DSL) library in CloudCity (written in Scala). These agent programs are then compiled into Scala, which can be executed directly by our engine. We also make queries over the outcome of a simulation easy by integrating simulations into data science pipelines via an analytical operator `Simulate`. Users can nest different simulations to form complex data science pipelines. In this section, we describe our DSL in detail and show how simulations can be composed with other data science pipeline components through `Simulate`.

2.1 Agent DSL

Users define agents by creating a subclass of `Agent` class defined in the CloudCity library. An *agent* is an object instantiated from a subclass of `Agent`. There is no concurrent read or write to an agent's attributes or methods. Agents are single-threaded during the execution of a simulation.

Agents communicate through messaging. Each agent has a unique id and a mailbox that buffers received messages. An agent sends a message `m` using

$$\texttt{send(rid : Long, m : Message) : Unit},$$

where `rid` is the id of receiver agent. `Message` class is also defined in our library and can be extended to create subclasses. Sending a message is fire-and-forget. To retrieve a message from the mailbox, an agent calls

$$\texttt{receive}() : \texttt{Option[Message]},$$

which returns `None` if mailbox is empty.

By default, messages take one round to arrive. Some messages can tolerate a longer delay. For example, an agent may process messages once every 20 rounds. Not all messages sent to this agent need to arrive in the next round. Our library defines `TimedMessage`, which is a subclass of `Message` with additional attributes `sentTime` and `delay`, in the unit of rounds. The attribute `sentTime` (i.e., the time of sending) is set by our system. A *timed message* is an instance of `TimedMessage` that arrives after `delay` rounds rather than one round.

In addition to supporting *message-passing protocol* using `send` and `receive`, our library also provides support for *remote procedure calls (RPC)*. Agents send RPC requests with

$$\texttt{asyncCall}(\texttt{receiver.API}(\texttt{args}^*) : \texttt{T}, \texttt{delay} : \texttt{Int}) : \texttt{Future[T]},$$

which returns a future object used by the sender to check the arrival of the RPC reply and to retrieve the RPC's return value. `Future[T]` is defined in our library, with similar usage to that in the standard Scala library. During code generation, our system compiles RPC calls down to message-passing, by creating corresponding RPC request messages that arrive in `delay` rounds.

If an agent does not want to receive a reply after sending an RPC request, then it can use

$$\texttt{callAndForget}(\texttt{receiver.API}(\texttt{args}^*) : \texttt{T}, \texttt{delay} : \texttt{Int}) : \texttt{Unit},$$

which has a signature similar to `asyncCall` but returns nothing. Unlike `asyncCall`, `callAndForget` does not generate a future object and thus has better performance. RPC request messages generated by `asyncCall` and `callAndForget` can be distinguished by a `sessionId` attribute, which is `None` for messages generated for `callAndForget`.

Using `receive` together with other computation instructions is one way to process RPC requests. Alternatively, a receiver can handle such requests in batches via

$$\texttt{handleRPC}() : \texttt{Unit},$$

which traverses received messages in an arbitrary order. For each RPC request, the receiver calls the corresponding method and sends a reply message when applicable. An RPC reply has the same `delay` and `sessionId` as the corresponding request.

An agent executes its instructions sequentially and synchronizes with other agents at the end of each round (explained in Section 3). Instructions are separated into different rounds via

$$\texttt{wait}(\texttt{n} : \texttt{Int}) : \texttt{Unit}.$$

Intuitively, in round t, an agent that executes `wait(n)` becomes idle until round $t + \texttt{n}$ starts, and only then it will resume executing instructions that follow `wait(n)`.

We illustrate how to define agents using these DSL instructions in Code 1, for a minimal epidemics example described below.

Example 2.1. We model the spread of a disease. For simplicity, we assume that a person is either healthy or infectious, represented by a Boolean attribute `infectious`. A healthy person becomes infectious when in contact with an infectious person. An infectious person remains infectious indefinitely. The disease spreads as infectious people randomly contact others.

Agent definition in Code 1 resembles a typical object-oriented program. Each person agent has attributes (lines 2–3) and methods (`infect`, `main`). The `infect` method (lines 5–7) can be specified in RPC requests for execution. On line 14, we use `callAndForget` to send RPC requests for `infect`. A healthy person becomes infectious after processing RPC requests.

```
 1  class Person extends Agent{
 2    var infectious: Boolean = Random.nextBoolean()
 3    var outNeighbors: Col[Person] = Col[Person]()
 4
 5    def infect(): Unit = {
 6      infectious = true
 7    }
 8
 9    def main(): Unit = {
10      while (true) {
11        handleRPC()
12        if (infectious) {
13          outNeighbors.filter(_=>Random.nextBoolean())
14            .foreach(i=>callAndForget(i.infect(), 1))
15        }
16        wait(1)
17      }
18    }
19  }
```

Code 1. Agent definition

The round-based behavior of a person agent is defined in `main` (lines 9–18): in every round, an agent processes all RPC requests using `handleRPC` (line 11). If infectious, an agent infects neighbors randomly selected from `outNeighbors` (lines 12–15). Instruction `wait(1)` (line 16) divides the computation of the `while(true)` loop (lines 10–17) into different rounds. Simulations only run for finitely many rounds, as specified by users (see below).

2.2 Simulate Operator

We add to the arsenal of data scientists a new operator `Simulate`, which enables simulations to interoperate with other components of a data science pipeline through the following interface

$$\texttt{Simulate(agents: Col[Agent], total: Int): Seq[Col[Agent]]},$$

where `Col`$[T]$ and `Seq`$[T]$ are collections and sequences of a type variable T respectively. `Simulate` is called with a collection of agents and the number of rounds `total` that a simulation should run for. This operator returns a time series, i.e., a sequence of snapshots, where each snapshot is a collection of agent objects taken at the end of each round.

In large-scale simulations, auxiliary simulations are common for reconstructing missing data or comparing different configurations. Such simulations form complex data science pipelines and can greatly benefit from composing simulations as an operator with other components in the pipeline.

Here we revisit Example 2.1 and present expressions that integrate simulations in data science pipelines. Researchers may be interested in questions similar to those in Table 1. `Simulate` operator makes it easy for users to express such questions, shown in Scala pseudocode. We assume that each round represents a day and a simulation runs for 90 rounds by default.

Code 2 shows how to compose different operators in the data science pipeline for Example 2.1, including preprocessing and postprocessing. The DSL instructions in the agent class definition in Code 1 need to be translated to Scala, which is described later in Section 5. To distinguish, we refer to the generated class definition for `Person` as `generated.Person` (line 4). The social network graph of the population is generated using the Erdős-Rényi model [28], omitted in Code 2. The probability that any two people are connected is assumed to be 0.01. For postprocessing, we select

Table 1. Sample questions and query expressions

Q1	How does the total number of infected population change over time? `Simulate(agents, 90).map(_.filter(_.asInstanceOf[Person].infectious).size)`
Q2	How many individuals are infected in the end, averaged over 10 simulations? `Range(1, 10).map(Simulate(agents, 90).last.filter(_.asInstanceOf[Person].infectious).size).average`
Q3	Start a new simulation from day 30 of an existing simulation. How many people are infected on day 30 of the new simulation? `Simulate(Simulate(agents, 30).last, 30).last.filter(_.asInstanceOf[Person].infectious).size`

```
object EpidemicSimulation {
  def apply(population: Int): Seq[Int] = {
    // Preprocess
    val people = (1 to population).map(new generated.Person)
    val agents = ErdosRenyiGraph(people, 0.01)
    // Start simulation
    val timeseries = Simulate(agents, 90)
    // Postprocess, query evaluation
    timeseries.map(_.filter(_.asInstanceOf[Person].infectious).size)
  }
}
```

Code 2. Full simulation pipeline

Q1 from Table 1. This query returns a sequence of integers, where each integer represents the number of infected people at the end of a round.

3 MODEL OF COMPUTATION

In the previous section, we stated that a simulation proceeds in a sequence of rounds, which is similar to the BSP model [67], where computations proceed in a sequence of supersteps. However, different subsets of processors in the BSP model cannot synchronize locally. Large-scale simulations often form hierarchies among agents, where a group of agents communicates more frequently and at lower latency than others. In this case, simulations can greatly benefit from supporting partitioned execution where different groups of agents synchronize locally for some time before synchronizing globally. Thus, a *hierarchical* BSP model [7, 9, 13, 57] may be more desirable. We start with making some basic notions precise.

Definition 3.1 (Agents). An *agent* is a sequential thread that communicates with other agents by sending messages. For this purpose, every agent has a *mailbox* that buffers incoming messages.

Each agent has an associated *logical clock* (initially 0) that changes in discrete ticks. Per agent, *round* t ends when its logical clock is incremented to $t + 1$, and execution begins with round 0. A round corresponds to a tick.

The agents' logical clocks are not general logical clocks but much more well-behaved: They are updated only through synchronization mechanisms in the system and cannot be otherwise manipulated, in particular not by the agents themselves. We describe how the system changes the clocks of agents later in this section. Depending on how the simulation is run (detailed below), the clock values between the agents need not be synchronized.

For ease of presentation, we explain the model in terms of the following instructions:

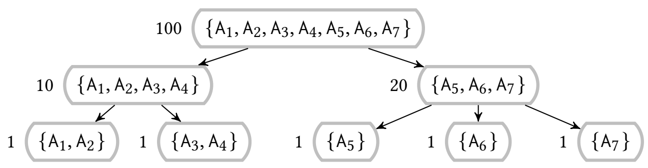


Fig. 1. A hierarchical partition of a set of 7 agents.

(1) `send(B, m, k)`: send message `m` to `B` that is delivered when `B`'s clock gets incremented to $t + \mathtt{k}$ where t is the time of `A` at the point of sending.
(2) `wait`: wait for the next synchronization (a syntactic sugar for `wait(1)`) – the agent can continue working after its clock has been incremented by one.
(3) `receive`: fetch an arbitrary message from the mailbox (or return nothing if the mailbox is empty).

This matches our description in Section 2 with the only difference that in the `send` instruction, we make the message latency `k` explicit and refer to a receiver agent `B` directly rather than by its id.

K-Availability. We say an agent `A` is *K-available*[3] at time t to agent `B` if messages from `B` to `A` sent in the interval $[t, t + K - 1]$ arrive at $t + K$. It follows that if `A` is K-available to `B` in round t for some $K > 1$, then *necessarily*, `A` is $(K - 1)$-available in round $t + 1$. Agent `A` can process a message `m` only if its clock has reached the expected arrival time.

Messages between different agents can be arbitrarily delayed, which can be viewed as the occurrence of a network partition that separates agents into different components. As an optimization, we can partition agents based on their availabilities.

In the following, we discuss two different semantics of a simulation, *non-partitioned* and *partitioned* execution. Non-partitioned execution directly corresponds to BSP in the sense that computation proceeds in "global rounds" where after every round, all agents are synchronized. Partitioned execution models what we call the *weighted hierarchical BSP* model.[4] Intuitively, we prescribe a recursive partition of the set $S = \{\mathsf{A}_1, \ldots, \mathsf{A}_n\}$ of all agents. Per set C in this recursive partition, we fix a separate number K, indicating that this set of agents is synchronized every K round.

Definition 3.2. A *weighted hierarchical partition* of S is a tree $\boldsymbol{C} = (V, E)$ with two vertex-labeling functions $C\colon V \longrightarrow 2^S$ and $K\colon V \longrightarrow \mathbb{N}$ such that

- for the root vertex r we have $C(r) = S$;
- for any inner vertex v of $\boldsymbol{C}$ with set of children N_v, we have
  (1) $|N_v| \geq 2$ and $\big(C(w)\big)_{w \in N_v}$ is a partition of $C(v)$ and
  (2) for every $w \in N_v$ there exists an integer $\alpha_w > 1$ such that $K(v) = \alpha_w K(w)$; and
- for any leaf vertex v, we have $K(v) = 1$.

We call the sets C for which there exists a vertex v in $\boldsymbol{C}$ such that $C = C(v)$ the *components* of $\boldsymbol{C}$. Note that every component appears as the label of exactly one vertex, allowing us to identify vertices and components. Hence, if $C = C(v)$ is a component and $w_1, \ldots, w_n$ are the children of v, then we let $K(C) = K(v)$ and call $C(w_1), \ldots, C(w_m)$ the children of C.

Example 3.3. Figure 1 depicts an example hierarchical partition of the agents $\mathsf{A}_1, \ldots, \mathsf{A}_7$. The labels indicate the parameters $K(C)$.

[3]This is different from K-safety, where a system has K replicas of data.
[4]See Section 8 for a brief discussion of related models.

Now in addition to the set of agents S, we also fix a *weighted hierarchical partition* $\boldsymbol{C}$ of S.

Condition 3.4. With fixing $\boldsymbol{C}$ we impose a restriction on the usage of `send` instructions in order to match the intuitive meaning of $\boldsymbol{C}$ we described before: If agent $\mathtt{A}$ executes $\mathtt{send}(\mathtt{B}, \mathtt{m}, \mathtt{k})$ in round r, then it is required that $\mathtt{k} \geq k_0$ where k_0 is the smallest positive integer such that $r + k_0 \equiv 0 \pmod{K(C(\mathtt{A}, \mathtt{B}))}$, where $C(\mathtt{A}, \mathtt{B})$ is the lowest common ancestor of $\mathtt{A}$ and $\mathtt{B}$.

We further assume that all messages sent during the simulation are unique. If $\mathtt{m}$ is a message that was sent by $\mathtt{A}$ using $\mathtt{send}(\mathtt{B}, \mathtt{m}, \mathtt{k})$, then we say $\mathtt{m}$ is *scheduled to arrive at* time $t + \mathtt{k}$ where t is the time of $\mathtt{A}$ at the point of sending. *In any case, if an agent* $\mathtt{A}$ *attempts to send a message violating Condition 3.4, an exception is thrown, and the simulation is aborted.*

Non-partitioned Execution. In this execution mode, we assume a separate *global leader* process that governs the synchronization. Whenever an agent calls `wait`, it notifies the leader, blocks, and waits for a response. The global leader, in turn, waits to get such notifications from all agents. Once they have all arrived, it sends out the responses. In this situation, all agents' clocks are incremented by one, and all pending messages get put into their destination mailbox. If t is the new clock value of $\mathtt{A}$, and $\mathtt{m}$ is a message in the mailbox of $\mathtt{A}$, then $\mathtt{m}$ remains locked if it is scheduled to arrive at some $t' > t$, and is unlocked otherwise. Receive instructions can only retrieve unlocked messages.

Partitioned Execution. For partitioned execution, synchronization is governed by *local leader* processes for the components.[5] Consider some clock value t. For every agent, there exists some component C in $\boldsymbol{C}$ that is closest to the root such that $t + 1 \equiv 0 \pmod{K(C)}$. This is the same component for all agents in C. At some point, all agents in C have their clock at t. Then the next synchronization of C is handled by a local leader for C. The interaction of the agents in C with this leader is precisely like described in the non-partitioned execution. In particular, when the leader has received notifications from all agents in C, the clocks of all agents in C get incremented to $t + 1$. At this point, all pending messages *within* C arrive at their target mailboxes, and use the same locking mechanism as in non-partitioned execution.

If C is a component in $\boldsymbol{C}$, then the agents in C get synchronized every $K(C)$ rounds of their local clocks. This means that their clocks are incremented together, and all pending messages between the agents of C arrive. In particular, for every $K(S)$ rounds, the full set of agents is synchronized.

Indistinguishability. Above, we have given two semantics for the execution of an agent-based simulation. In either case, the clock of every agent gets updated in increments of one. Therefore, the time series of the simulation can be defined as the sequence $q_0, q_1, q_2, \ldots$ where every q_i contains a snapshot of all agents' states and their mailboxes right before the agents' clock got incremented to i, and where q_0 contains all the initial states.

It is now easy to show that (subject to Condition 3.4) the behavior of the simulation is not affected by the mode of execution.

Proposition 3.5. *Let* $\boldsymbol{C}$ *be a hierarchical partition of the set of agents* S. *Then every time series of* S *under non-partitioned execution is a time series under partitioned execution, and vice versa.*

Proof. Suppose we have a partitioned execution producing a particular time series. As enforced by the semantics, between any two subsequent synchronizations of a component C, the same number of rounds have passed for all agents in C. Converting the local synchronizations (incrementing t to $t + 1$) into global ones by aligning the points of synchronization does not alter any computation or communication. Messages that arrive earlier in the target mailbox due to this change still remain

[5]The leader for the root component is the global leader from before.

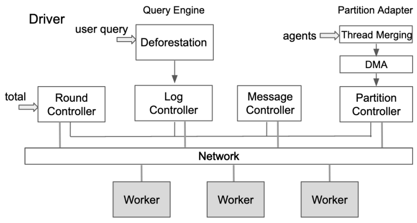


Fig. 2. System architecture

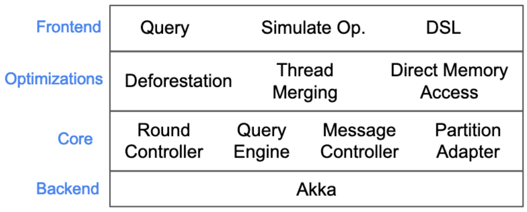


Fig. 3. Software stack

locked until their scheduled arrival time. Applying this to all synchronizations yields an execution in the non-partitioned mode that produces the same time series.

For a non-partitioned execution, the global synchronizations can likewise be turned into local ones (according to $\boldsymbol{C}$). Hence, some messages will reach their target mailbox later. Because Condition 3.4 needs to be satisfied, this poses no problem though, as the next synchronization where they can arrive is at most their scheduled arrival time. Therefore, this again does not affect the computation or communication, and yields a partitioned execution producing the same time series. □

Our model (and the result above) can easily be extended to allow dynamic partitions in the following sense. At any time value t, at which there is a synchronization of a component C, the subtree rooted at C can be replaced with a different hierarchical partition of C. This requires that the new value K' for $K(C)$ is compatible with the number $K(C')$ of the parent component C' of C, taking into account how many rounds until the next synchronization of C' have already passed. That is, formally, K' needs to divide both $K(C')$ and $K(C') - t \bmod K(C')$.

4 SYSTEM ARCHITECTURE

CloudCity is a large-scale agent-based simulation framework that is based on our BSP-like computational model. We would like to combine the best of both worlds, maintaining the flexibility of agent-based simulation frameworks for describing complex simulations while achieving the efficiency of parallel computing in BSP-like systems. Our distributed simulation engine is built around the weighted hierarchical BSP model described in Section 3, with an overall structure that follows the driver-worker architecture: Driver evaluates user programs and coordinates workers; each worker has a subset of agents, which we refer to as its *local agents*.

Driver-worker architecture naturally forms a two-level hierarchy. Driver corresponds to the global leader in Section 3. For partitioned execution, the driver synchronizes with all workers every $K(S)$ rounds. Recall that when calling `Simulate`, a total number of rounds (`total`) is specified. If $K(S)$ exceeds the number of remaining rounds, then the driver synchronizes workers when `total` rounds have been reached. For simplicity, we assume that `total` is a multiple of $K(S)$ in this section. Workers have recursive components according to a weighted hierarchical partition of S. Each component C has a local leader that synchronizes its local agents every $K(C)$ rounds. Non-partitioned execution is a special case when the system has only one component and $K(S) = 1$.

Figure 2 illustrates our system architecture, with applicable optimizations. Below, we describe the driver modules: round controller, query engine, message controller, and partition adapter.

Round Controller. At the beginning of every $K(S)$ round, the round controller instructs each worker to execute $K(S)$ rounds and waits for their completion. Each worker informs the driver when it finishes. If `total` rounds have passed, then the round controller terminates the simulation.

Query Engine. The query engine evaluates user queries over the time series. If there is no user query, then this unit outputs the time series when the simulation ends. After executing $K(S)$ rounds, each worker forwards the log of its local agents over the past $K(S)$ rounds to the log controller of the driver, which aggregates the logs from all workers. When applicable, the query engine performs deforestation (see Section 6) before materializing the data for query evaluation.

Message Controller. In our system, an agent can start executing a round only after receiving all messages that should have arrived. The message controller provides such information to agents through *expect-from* lists. The synchronization of local agents in workers is described in Section 3. At the end of $K(S)$ rounds, each worker collects messages from its local agents that are sent to other workers and informs the driver which workers it will send messages to via a *send-to* list.

The message controller processes the send-to lists from all workers and generates an expect-from list for each worker. Upon receiving the expect-from list, each worker sends messages to other workers in batch and waits for messages from workers on the expect-from list before resuming the local agents. Additionally, communication profiling can be enabled in the message controller, to provide message analyses per worker.

Partition Adapter. The partition adapter separates agents into components, as described in Section 3. Per component, it may de-parallelize agent threads to reduce resource contentions using thread merging and apply locality-based optimizations, such as direct memory accesses (see Section 6). The partition strategy for separating agents into components can be static or dynamic: statically, agents are separated according to a fixed weighted hierarchical partition; dynamically, the partition adapter re-partitions agents based on the message analysis received from workers.

Figure 3 summarizes the software stack of CloudCity. The *core* refers to the distributed engine. The programming model forms the *frontend*. The DSL instructions in the agent class definitions are compiled to Scala during the code generation, which we explain next. The *optimizations* layer summarizes applicable system optimizations for the distributed engine (see Section 6). The *backend* of our system is built using a Scala library Akka [2] to handle distributed networking.

5 CODE GENERATION

We translate DSL instructions to Scala using multi-stage programming [55, 56]. Here we focus on generating instructions for `wait(n)`; other DSL instructions are straightforward.

Recall from Section 2 that an agent executing `wait(n)` in round t waits until round $t + \texttt{n}$ starts. Conceptually, when executing `wait(n)`, an agent proposes the number of rounds that it is waiting for (referred to as `proposeInterval`) to the local leader of the component. After receiving such values from all agents in the component, the local leader increments the logical clock of the agents by the minimum of the proposed values and resumes the agents. The agent checks the logical clock and repeats the process until round $t + \texttt{n}$.

We implement `wait(n)` using coroutine instructions [22]. The control transfer between a local leader and an agent can be modeled using `run` method[6] and a Boolean variable `yieldFlag` (initially false). A local leader *resumes* an agent by calling `run` method defined in an agent; an agent *yields* control to the local leader implicitly when `run` terminates and returns. Instruction `wait(n)` changes `proposeInterval` (as we have described above) and sets `yieldFlag` to `true`, which causes `run` to stop and return (see Code 3).

The method `run` is similar to `Compute` in Pregel [45]; both serve as the API to interact with a system at each round. However, in Pregel, users need to explicitly separate computations in a vertex

[6]This can be viewed as a single control-transfer instruction in a symmetric coroutine [22].

```
val instructions = Array[() => Unit]()
var nextInst: Int = 0
// wait(n) changes proposeInterval and yieldFlag
var proposeInterval: Int = 1
var yieldFlag: Boolean = false

def run(): Int = {
  yieldFlag = false
  while(!yieldFlag){
    instructions(nextInst)()
  }
  proposeInterval
}
```

Code 3. A snippet of the generated program in CloudCity

to different supersteps for resuming computation across different supersteps. In CloudCity, the system automatically determines how computation should be divided into different rounds based on `wait(n)`, which improves the usability of our programming model.

6 OPTIMIZATIONS

Our system optimizations focus on decreasing the degree of concurrency through thread merging, compiling messages to direct memory accesses, and reducing intermediate data when evaluating user queries through deforestation, described below.

Thread Merging. By default, every agent in our system is a thread. As the number of agents increases, the degree of concurrency in the system grows, which worsens resource contention and leads to inefficient hardware utilization. We address this by sharing a thread among multiple agents through thread merging. We refer to such threads and agents as *merged threads* and *merged agents* respectively. When a set of agents is merged, each of the agents is transformed into a coroutine instance. These merged agents know the ids of each other, which can be exploited for further optimizations, such as direct memory accesses (see below). The merged thread encapsulates the coroutine instances and executes them sequentially in every round.

Direct Memory Accesses. Messaging comes with the computation overhead of packing and unpacking messages and the memory overhead of extra buffering, which can be avoided if agents call the methods in the receiver directly, which we refer to as direct memory accesses (DMA).[7] The caveat is that DMAs cause *immediate* state changes. Under the messaging semantics, an agent executes each method specified in messages atomically. The atomicity property can be violated if concurrent agents call the same method via DMA simultaneously. To address this, our system applies DMA only when senders and receivers share the same thread, thus no concurrent invocations will occur. Still, DMA can change event orderings, which may cause unexpected behavior in the receiver, see the example below.

Example 6.1. Consider three agents, A_1, A_2, and A_3, that are merged together in this order. A_2 models a shared counter with a variable `counter` (initially 0) and an operation `inc`. In every round, A_1 and A_3 send an RPC request that calls `inc` (with delay 1) in A_2. A_2 processes all messages and then prints `counter` (see Figure 4a).

[7]We overload the notion of direct memory accesses, which typically refers to a technique used by devices to transfer data to or from memory without CPU involvement, enabling high-speed data transfer and reducing CPU utilization.

```
 1  var counter:Int = 0
 2  def inc():Unit={
 3    counter += 1
 4  }
 5  def main(): Unit ={
 6    while (true) {
 7      handleRPC()
 8      print(counter)
 9      wait(1)
10    }
11  }
```

(a) No DMA

```
1  val counter = Map[Int,
       Int](0 -> 0)
2
3  @allowDirectAccess
4  def inc(v: Int): Unit = {
5    counter(v) = 1 +
         counter.getOrElse(v,
          counter(v-1))
6  }
7
8  // print(counter(time))
```

(b) Multi-version DMA

```
 1  var g: Int = 0
 2  var counter: Int = 0
 3  @allowDirectAccess
 4  def inc(): Unit = {
 5   if (agentInactive) g+=1
 6   else counter += 1
 7  }
 8  def resolve(): Unit = {
 9    counter += g
10    g = 0
11  }
```

(c) Generalized double-buffering DMA

Fig. 4. Variants of A_2 definition for Example 6.1

With DMA, A_1 calls `inc` and modifies the value of `counter` before A_2 runs; in the first round, A_2 prints 1. With messaging, no message arrives in the first round and A_2 prints 0. Hence users can distinguish whether DMA has been applied.

We can separate different effects via multi-versioning, as shown in Figure 4b. The `print` instruction (line 8) is applied to the version of `counter` that is visible in the current round. The `inc` operation takes a version number as a parameter (line 4) and updates the value for the given version (line 5). If the value for a version v does not exist, then it is created and initialized with the value for the previous version $v - 1$. In this example, agents in round t call `inc` with version key $t + 1$, thus delaying the immediate effect of DMA to a future version of `counter`.

Line 3 in Figure 4b contains a new annotation `@allowDirectAccess`. For now, DMA needs to be enabled explicitly using this annotation. For

$$\texttt{callAndForget(receiver.API(args}^*\texttt{), latency)} : \texttt{Unit},$$

the sender calls `receiver.API` directly if it shares the same thread with the receiver and `receiver.API` allows such accesses.[8]

In this example, all messages arrive in one round, hence only two versions are necessary. We can use generalized double-buffering in such a way that calling `inc` via DMA changes the value of an auxiliary variable `g` instead of `counter`, see Figure 4c. A_2 still prints the value of `counter`. At the beginning of each round, the system calls `resolve` to merge `g` and `counter` (lines 9–12). The Boolean variable `agentInactive` (line 6) is `false` when an agent is running and `true` otherwise.

Deforestation. The `Simulate` operator generates a time series in the form of a sequence of snapshots of all agents (see Section 2). Although this rich form of time series lets a user compose and nest different simulations, such flexibility comes at the cost of generating a large amount of intermediate data and is not always necessary. If a user is only interested in evaluating a single query over the time series and composes this query expression with `Simulate` before starting to execute a simulation, we can apply deforestation [68] to materialize only data pertinent to the user query to reduce such costs.

To illustrate, we revisit Q1 in Table 1, which concerns how the infectious population changes as the disease progresses. We assume non-partitioned execution mode. Generalizing the technique presented here to partitioned executions is straightforward.

[8]We only consider `callAndForget` because this instruction does not return any value.

At the end of each round, a worker applies a function `m`

```
m: Col[Agent] => Int = (agents: Col[Agent]) =>
    agents.filter(_.asInstanceOf[Person].infectious).size,
```

to project the agent states only to the measure of interest, which reduces the log size before being forwarded to the query engine (see Section 4). The log controller in the query engine maintains a partial result for Q1, a sequence of the total number of infectious people in each round so far. The new log received from all workers in this round is reduced by the log controller via function `r` before being appended to the partial result,

```
r: Col[Int] => Int = (logs : Col[Int]) => logs.sum.
```

7 EXPERIMENTS

In this section, we assess the performance of our system empirically. While we also examined existing general-purpose distributed agent-based simulation engines [17–19], such engines do not support efficient parallel computation or leave the problem of parallelization to the user (see Section 8). Given the synchronous round-based simulation semantics that we use in this paper, we evaluate our system by comparing it with state-of-the-art distributed systems that support BSP-like computation: Spark [72], GraphX [37], Giraph [3, 16], and Flink Gelly [12, 33]. We select examples from representative fields to serve as the basis for comparing different systems. We use a cluster of servers for all experiments. Each server has 24 cores (two Intel Xeon E5-2680, 48 hardware threads), 256GB of RAM, and 400GB of SSD.

7.1 Workloads

We consider classic examples from areas where simulations are prevalent: population dynamics, economics, and epidemiology.

Population Dynamics. We simulate the dynamics of a population modeled after Conway's Game of Life [35]. There is only one agent type. The agents are arranged in a 2D torus, i.e., each agent is connected to eight neighbors. The update rules follow [35]. Per round, agents process all received messages and send one message to all neighbors. Messages contain a single value that encodes whether an agent is alive and arrive at neighbors in one round.

Economics. We model an evolutionary stock market where the share price of a stock is seen as an emergent property of the buy or sell behavior of traders [54]. The stock market has one stock. The dividend per share is modeled as a stochastic variable. Both traders and stock markets are agents. For simplicity, we assume a single market agent, which communicates with all trader agents.

In each round, the stock market notifies all traders of the latest share price, dividend per share, and a list of conditions. A condition describes an event that a trader takes into account when making buy or sell decisions, such as whether the average share price over the last 50 rounds has increased or whether the dividend per share has increased.

Our model is a simplified version of the simulation described in [54]. We consider only five conditions, which can be expanded to make the simulation results more realistic. Traders are rule-based systems that aim to maximize their wealth, which consists of bank savings and stock shares.[9] A condition-action rule of a trader decides whether the trader should buy or sell a stock share, based on conditions received from the market. Rules are ranked according to their strength, initially zero. If the wealth of a trader has increased after applying a rule, then the strength of this

[9]See [54] for update rules for the trader's wealth.

rule increases by one; otherwise, it decreases by one. In each round, traders act according to the strongest rule for the given condition.

Epidemiology. We have seen a minimal epidemics example in Section 2. Here, we extend the classic SIRD model with two extra states, E and H. More specifically, a person is in one of the following states: susceptible (S), exposed (E), infectious (I), recovered (R), hospitalized (H), or deceased (D). A susceptible person may become exposed only when in contact with a person in state I or H. The exposed state models the incubation period of the disease: A person in state E is infected but not yet infectious. That is, such a person cannot make others sick. A recovered person remains immune. The hospitalized state models how long a patient stays in a hospital, which can estimate whether medical resources become overloaded. Initially, 1% of the population is in state I, and the rest are in S.

A person may be isolated due to quarantine or lockdown policy, waiting idly for days before contacting others. The transition between different states (except from S to E, which requires communication between agents) is probabilistic, subject to how vulnerable a person is. We assume that the vulnerability depends only on age. Senior people in state I are more likely to transition into state H or D than young people. The duration of different states is modeled after the corresponding statistics for Covid-19 [40].

The connectivity of the population in a region is described by a social network graph. Each vertex represents a person. We say that a person A can contact person B if there is an edge between A and B in the social network graph. Individual infectiousness when contacting a neighbor is randomly generated from a gamma distribution with mean 1 and a shape parameter 0.25 [40]. The connectivity between people in different regions can also be modeled by a social network graph. We consider the following two graph models.

The Erdős–Rényi model (ERM) generates a random graph with n vertices, where an edge (i, j), $i \neq j$ is present with probability p [8]. All edges are independent. Another random graph model is the stochastic block model [39] (SBM), which generates a graph with communities. The set of vertices in a graph is partitioned into non-empty, disjoint blocks. The SBM is characterized by parameters p and q. For each pair of distinct vertices (i, j), if they are in the same block, then the probability that there is an edge between i and j is p. Otherwise, the probability is q.

In the implementation used for cross-comparing different systems, we consider only people agents. The social network graph is generated by ERM ($p = 0.01$) or SBM (5 blocks, $p = 0.01$, $q = 0$). We use the SBM just to have a clear block structure and set $q = 0$ for simplicity. An in-depth evaluation of the workloads, such as how the value of q changes the performance, is not our focus. In a more complex extension, we include hospitals and governments (equivalently, regions) as agents. A hospital accepts or rejects patients based on available beds and reports to the government periodically about its capacity. The government adjusts the policy accordingly: If hospitals reach 80% of capacity, then the government issues a lockdown; if hospitals are 50% full, then the infected people need to quarantine. The policy is also affected by other governments. If half of its neighboring governments adopt a policy, then the government follows suit. The social network graph describes how the residents are connected, both within the same region and across regions. The government informs its residents of any policy changes.

7.2 Implementation Details

We implement the workloads such that the agent code in CloudCity or other systems, including Spark, Giraph, GraphX, and Flink Gelly, has the exact same behavior. Since the programming models of these systems vary, we explain important details for different implementations. In Giraph, GraphX, and Flink Gelly, messages arrive in one superstep, vertices cannot make RPCs, and no time series is generated. We impose these restrictions consistently across all systems, which require that all agents in CloudCity are exactly 1-available to others in every round, agents do not use RPC instructions, and the log controller is disabled. Hence, system optimizations including direct memory accesses and deforestation that depend on such functionalities are inapplicable.

The programming model of Spark is RDD-based, where an RDD (Resilient Distributed Dataset) is an immutable collection of objects that can be parallelized [72]. For stateful objects with mutable states that need to be persisted even after a method returns, RDDs capture the serialized representation of such objects and store any operations over these objects in the lineage for lazy evaluation [72]. If the state of an object changes, upon materialization, Spark creates a new RDD where the modified state has been updated, and all other objects in the previous RDD are copied. While it is possible to transform a collection of agents as defined in our DSL directly to an RDD in Spark, our earlier experiments (not included in this paper) showed that changing agents to stateless is orders of magnitude faster due to reduced object creation and copying in Spark. In the stateless implementation, the "state" of an agent is captured in a data structure and is updated as a simulation progresses through the following APIs: `run` method consumes this data structure and a collection of incoming messages, and returns the updated agent state after processing the messages; `sendMessage` method consumes the state of an agent and returns one or more generated messages that will be sent to all connected agents. We adopt stateless agents for the Spark implementation when compare with other systems.

Giraph is an open-source implementation of the graph processing system Pregel [45]. Users define a graph algorithm in Giraph as a vertex program with a `compute` method, which is executed by all vertices in an input graph iteratively per superstep, as described in the BSP model. GraphX is a graph library for Spark and introduces optimized graph-specific operators. Gelly is a graph library of Flink [33], which provides a unified programming model for both batch and stream processing [12]. Both GraphX and Gelly support Pregel-like operators for iterative computation, which we employed in our experiments.

While Giraph, GraphX, and Flink Gelly are designed for graph processing, it is still feasible to implement simulations as vertex programs in these systems, but with some restrictions, such as the lack of support for polymorphic agents and messages. A vertex also turns inactive if it does not receive any message, which is problematic for simulations where agents often remain active to perform local computation besides handling messages. This issue can be addressed by introducing a clock vertex that sends heartbeat messages, which requires changes to both the input graph and the vertex program. Together, these limitations demonstrate that programming simulations as a vertex program in a graph system can be cumbersome and inefficient, highlighting why agent-based simulations call for a new system.

7.3 System Comparison

Table 2 summarizes system features that we believe are most relevant to efficient executions of agent-based simulations, which are stateful, communication-intensive, and exhibit high heterogeneity in agent code. We provide insights into the suitability of CloudCity and other state-of-the-art BSP-like systems for simulations, by analyzing the applicability of the features listed in Table 2 for each system. We use "x" to highlight systems with a given feature and explain them in detail below.

Systems that perform *in-place updates* directly modify the states of an object, including Giraph and CloudCity, allowing fast updates with no buffering overhead. In contrast, RDDs in Spark and GraphX are immutable; to perform computation over agents, as well as to perform the reshuffling of messages from agent outboxes to agent inboxes, multiple large collections of objects need to be maintained and joined. While Spark's lazy computation and internal deforestation optimizations reduce the number of copies held in memory, and explicit caching and unpersisting help make

Table 2. Features of different systems

	CloudCity	Giraph	Gelly	Spark	GraphX
in-place updates	x	x			
efficient local messaging	x	x	x		
asynchronous message reshuffling	x	x			
polymorphism	x				

memory management and garbage collection more efficient, Spark and GraphX still require the creation and copying of far more objects than systems that take a stateful BSP approach with mutable objects by supporting in-place updates. In Flink Gelly, in each iteration, vertices that need to be modified are duplicated and changes are applied to the duplicates rather than in situ.

Efficient local messaging refers to an optimization when agents are on the same machine. In CloudCity and Giraph, for instance, such messages are delivered by passing references, rather than copying and serializing message objects. In Spark, GraphX, and Flink Gelly, messages are delivered using a join-like operation that shuffles messages from agent outboxes to agent inboxes, without special treatment for messages that are to be delivered locally, between agents on the same machine.

Asynchronous message reshuffling allows agents to start executing the next round as soon as their mailbox has been filled in, which is a BSP-specific optimization that is supported in both Giraph and CloudCity. In contrast, agents in other systems begin executing the next round only after all the messages in the system have arrived.

Polymorphism refers to supporting multiple agent and message types. For instance, if a simulation contains different agent types, then polymorphic agents ensure that every agent contains only relevant attributes, which is more memory efficient than mixing attributes of all agent types. The performance advantage of polymorphic messages is similar. CloudCity is the only system that supports polymorphic agents and messages in a single simulation.

7.4 Evaluation

We examine the relative performance of CloudCity compared to other state-of-the-art BSP-like systems using the workloads described in Section 7.1. We begin with finding the best setup configuration, such as the number of workers per machine, of each system for later experiments. Next, we compare the scalability of the systems as the number of agents (scale-up) or machines (scale-out) increases. In addition, we stress-test key performance factors, like communication frequency and computation interval, for each workload. For CloudCity, we evaluate the effectiveness of different optimizations described in Section 6 and the performance impact of hierarchical communication patterns with delayed messages separately.

In the experiments, we simulate each workload for a fixed number of rounds, and we compute the average time consumption per round. For the population dynamics and economics examples, we use 200 rounds. For the epidemics examples, only infectious agents send out messages. Initially, the infectious population (and, therefore, the number of messages) grows exponentially, but at some point, it will drop to zero. To capture the "interesting" behavior when the disease spreads, the epidemics simulation is only run for 50 rounds. Each experiment is repeated three times and we show the average time per round, averaged again over all repetitions.

Configuration tuning. Our first goal is to find the best configuration for each of the systems. For this, we assume 10,000 agents per machine and evaluate the performance of different configurations

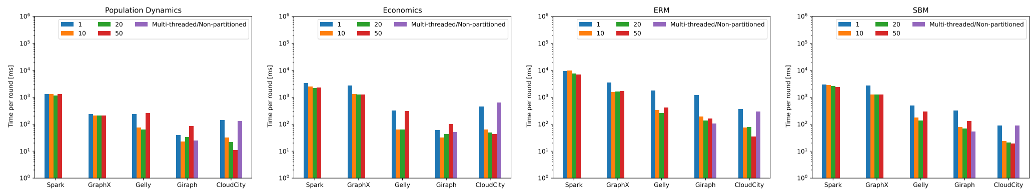
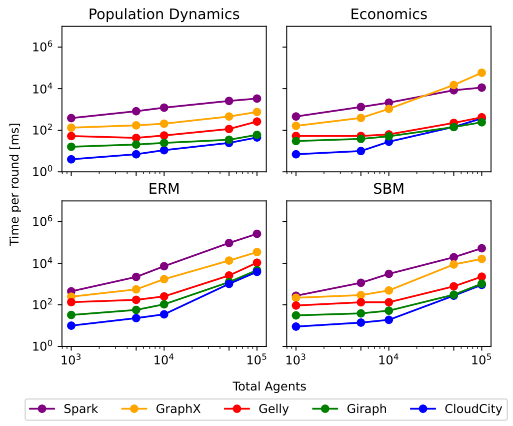
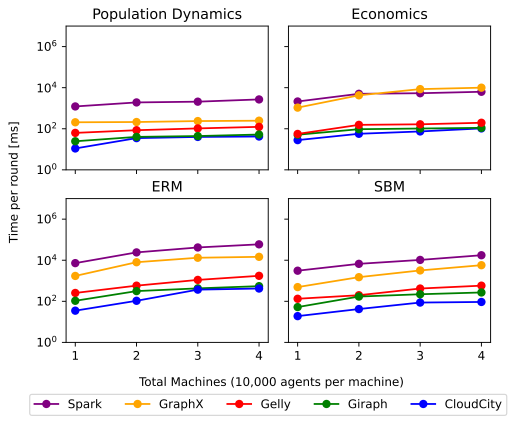


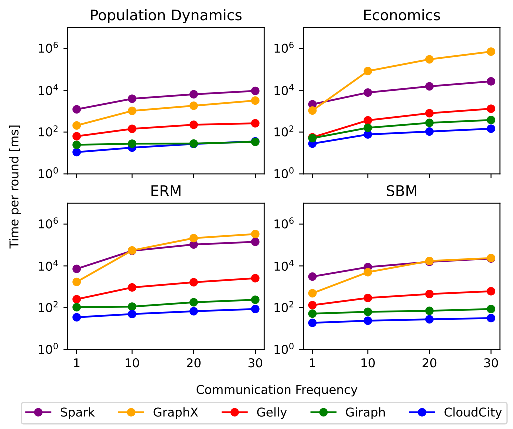
Fig. 5. Configuration tuning (system-specific unit of parallelism)



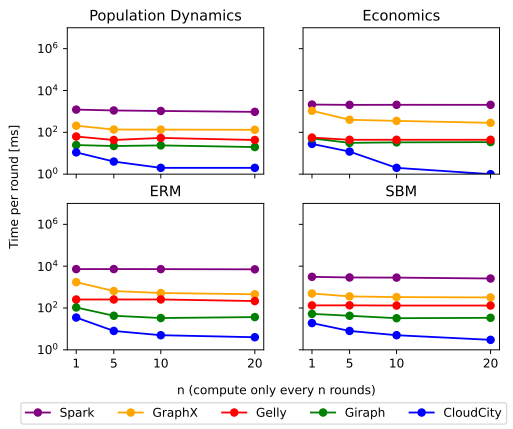
(a) Scalability (scale-up)



(b) Scalability (scale-out)



(c) Communication frequency



(d) Compute only every n rounds

Fig. 6. Performance evaluation (lower is better). (a) Increasing the number of agents; (b) Increasing the number of machines (10,000 agents per machine); (c) Microbenchmark: Increasing communication frequency; (d) Microbenchmark (idle periods): Increasing the number of rounds until the computation is resumed

individually per system. The terminology for achieving parallelism is different in these systems, which can be the number of cores (Spark, GraphX), parallel instances (Flink Gelly), workers (Giraph), or components (CloudCity), hence we omit the unit of the parallelism in Figure 5. The resulting best configurations are used in the subsequent experiments to cross-compare the performance across systems.

We tune both Spark and GraphX by increasing the number of logical cores. The cluster manager of Spark deals with the low-level resource allocation in a (possibly distributed) system. We employ the Standalone manager, which uses one multi-threaded Spark executor per machine [4]. Figure 5 show that increasing the number of logical cores for the Spark executor improves the overall performance of Spark and GraphX for all workloads, as more hardware parallelism is exploited. Both 20 and 50 cores outperform 1 or 10 cores, but the performance difference between 20 and 50 cores is not significant in Figure 5, and we select 50 cores as the default setup for later experiments.

For Giraph, we compare the performance when changing the number of workers per machine. We increase the number of workers from 1 to 50 for single-threaded workers. We also consider using one multi-threaded worker, which can use up to 50 compute threads. Figure 5 shows that one multi-threaded worker per machine has the best overall performance across different workloads, as also reported in previous work [16].

In Flink Gelly, a program consists of multiple tasks, where each task is split into several parallel instances for execution. Every parallel instance processes a subset of the task's input data. The number of parallel instances of a task is called its parallelism. We increase the parallelism from 1 to 50. Figure 5 shows that Flink Gelly achieves the best performance when the parallelism is 20.

For CloudCity, we compare non-partitioned execution against partitioned execution with a varying number of components. In the non-partitioned execution, each agent notifies the driver that it has completed at the end of every round. In the partitioned execution, agents are separated into components and notify the local leader of the component when they have finished. Only local leaders communicate with the driver. For the experiments, we use a two-level hierarchical partition, separating agents evenly (based on their ids) into different components, and applying thread merging to each component. The number of components is increased from 1 to 50.

Figure 5 shows that the partitioned execution with 50 components is 4-15× faster than the non-partitioned execution, even though all agents in CloudCity are configured to be 1-available (Section 7.2) and synchronize every round in both executions. There are several reasons for the speedup. Firstly, the driver processes messages sequentially. In the partitioned execution, synchronization messages sent from agents are processed in parallel by local leaders, and the driver only needs to process synchronization messages from local leaders, which is much more efficient than in non-partitioned execution, where the driver processes messages from all agents. Secondly, our system compiles away synchronization messages between a worker and its local agents in partitioned execution. Instead of sending a synchronization message, an agent yields the control back to the local leader when it completes execution, thus the local leader does not need to process synchronization messages. This allows our system to further reduce the synchronization cost.

For the following experiments, we configured each system according to the findings above. Per machine, we use one multi-threaded worker for Giraph; a multi-threaded Spark executor using 50 logical cores for Spark and GraphX; 20 parallel instances in Flink Gelly; and partitioned execution with 50 components for CloudCity.

Scalability. In this experiment, we compare the scalability of these systems by increasing the number of agents (Figure 6a) from 1,000 to 100,000, and the number of machines from 1 to 4 (Figure 6b). When scaling out, we fix 10,000 agents per machine; moreover, for the SBM epidemics specifically, we adjust the number of blocks of the SBM to match the number of machines. The experimental data shows that CloudCity achieves on-par or better performance than Giraph, Flink Gelly, Spark, and GraphX.

Giraph has a performance comparable to CloudCity when increasing the number of agents and machines. Supporting polymorphism does give CloudCity a slight performance advantage over Giraph. In the SBM experiment (Figure 6b bottom right), CloudCity can be 2× faster than Giraph when scaling out, because CloudCity partitions agents based on their communication pattern, which allows our system to reduce the number of cross-machine messages to zero in the SBM experiment. Flink Gelly is around 2× slower than CloudCity as the number of agents and machines grows, because CloudCity supports in-place updates of objects and asynchronous message reshuffling, as shown in Table 2. Spark and GraphX are one to two orders of magnitude slower than CloudCity, due to the lack of in-place updates and inefficient local messaging.

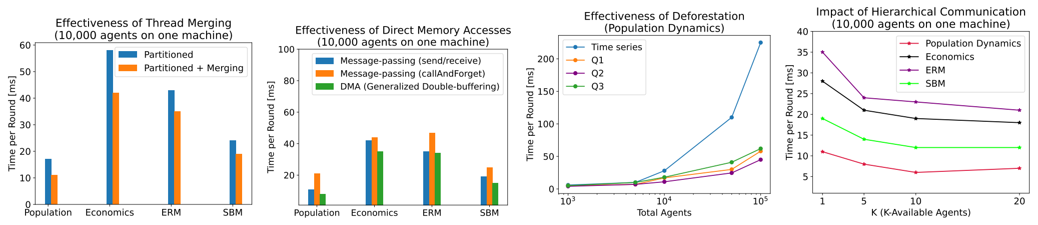
Our Spark implementation is 2-20× slower than GraphX as the number of agents and machines increases in population dynamics and epidemics examples, but can be faster than GraphX in the stock market example. The Spark implementation contains three RDDs: an immutable edge RDD that represents the input graph, a message RDD that contains a collection of all messages, and an agent RDD that contains a collection of agent states. Per round, the message RDD is joined with the agent RDD to deliver messages between agents. Afterward, agents proceed one round by executing `run` and `sendMessage`. The agent RDD is then materialized and joined with the edge RDD to produce new messages, which in turn updates the message RDD before the next round.

GraphX also contains three RDDs: vertex RDD (i.e. agent RDD), edge RDD, and triplet RDD, where the triplet RDD logically joins the vertex RDD and the edge RDD, connecting a source vertex with its destination vertices. The triplet RDD allows GraphX to generate and deliver messages more efficiently [37] than the Spark implementation. But the triplet RDD has a hidden cost: Vertices are duplicated; synchronizing these vertex copies requires much more object copying and data reshuffling than our Spark implementation.

Communication frequency. A key feature of simulations is that agents communicate frequently, thus we examine how different systems perform as the number of messages increases. For each workload in Section 7.1, we artificially increase the communication frequency by up to 30×, compared with the original workload. This is done by sending that many identical copies of every message. *These and all later experiments assume 10,000 agents on one machine.*

As we increase the number of messages to 30×, Figure 6c shows that the relative performance of CloudCity, Giraph, and Flink Gelly is approximately the same as in the scalability experiments, but Spark scales considerably better than GraphX. In the economics example, Spark is over two orders of magnitude faster than GraphX when the communication frequency is 30. Though both Spark and GraphX are RDD-based, the triplet RDD abstraction of GraphX can cause much more object copying and data reshuffling than our Spark implementation because of the duplicated vertices and messages, as we have described in the scalability experiment.

Idle periods. Another distinct feature of simulations is the presence of idle periods. For example, the epidemics simulation can include lockdown or quarantine policies, which prompt agents to wait idly for some rounds. We modify the workloads in Section 7.1 to microbenchmark this computation pattern, such that agents only communicate every n rounds, where $n = 1$ is the base case (no change from before), shown in Figure 6d. As expected, reducing the computation such that agents compute only every 20 rounds leads to lower execution time than computing every round for all systems. While CloudCity allows users to describe such computations easily using `wait(`n`)`, users of graph systems have to modify the input graph and the vertex program with an extra clock vertex, as explained in Section 7.2. The lack of support for expressing idle periods in such systems also has a performance cost. For instance, CloudCity is 10× faster than Giraph when $n = 20$, but the performance of these two systems is on par when $n = 1$.



(a) Thread merging (b) DMAs (c) Deforestation (d) K-availability

Fig. 7. CloudCity optimization evaluation

Thread merging. We evaluate the effectiveness of thread merging across all workloads. Same as before, we separate agents evenly into 50 components. *From now on, we assess system optimizations available only in CloudCity. For this, and the remaining experiments in this section, we only evaluate CloudCity on one machine with 10,000 agents, unless specified otherwise.* Figure 7a shows that merging each component leads to up to 2× improvement. The number of concurrent agents in each component far exceeds the number of hardware threads, which causes frequent context switches and worsens locality, both leading to sub-optimal performance. These issues are addressed by thread merging, which lowers the number of context switches and the number of branch misses.

Direct memory accesses (DMA). We estimate the effectiveness of DMAs in eliminating the overhead of message passing. We adjust the previous implementation of each example so that agents communicate using `callAndForget`, and design the RPCs in a way that allows direct memory accesses. The changes to the implementations for supporting the DMA require some events that occur when using message-passing primitives to be reordered. Users should be cautious and determine whether the behavioral changes caused by the event reordering are acceptable.

Figure 7b summarizes the evaluation of this optimization. The blue bar (left) denotes the performance of the implementation from before that uses `send`. The orange bar (middle) represents the performance of the implementation using `callAndForget`, which is up to 2× slower than `send` due to RPC-related instruction and memory overhead.

The green bar in Figure 7b illustrates the performance of the system when transforming `callAndForget` to DMAs, using generalized double-buffering. This optimization has eliminated the overhead of RPCs, achieving on par or slightly better performance than `send`. We also experimented with the performance of direct memory accesses implemented using multi-version, but it was 2-3× worse than generalized double-buffering, hence omitted from the comparison. The performance difference between generalized double-buffering and multi-version is due to memory activities. While generalized double-buffering pre-allocates memory needed for the auxiliary attributes, multi-version requires the system to dynamically allocate more memory for new versions of an attribute, and the memory occupied by old versions is not immediately freed.

Deforestation. We assess the effectiveness of deforestation using the population dynamics example as a case study and consider multiple queries, listed in Table 3. In the base case, time series experiments save a copy of each agent object at the end of every round in memory to generate the time series and return this time series to the user, allowing users to perform arbitrary queries over the time series. If users are only interested in a single query and compose this query with the `Simulate` operator when starting a simulation, then we can apply deforestation to save only data pertinent to the user query in the time series, which reduces the volume of the intermediate data.

Table 3. Query expressions for deforestation experiment

Q1	How many agents are alive during the simulation?
	`Simulate(agents, 200).map(x => x.filter(_.asInstanceOf[Cell].alive).size)`
Q2	How many agents are alive at the end of the simulation?
	`Simulate(agents, 200).last.filter(_.asInstanceOf[Cell].alive).size`
Q3	Which agents are alive in each round during the simulation?
	`Simulate(agents, 200).map(x => x.filter(_.asInstanceOf[Cell].alive))`

Figure 7c shows the average time per round when increasing the number of agents from 1,000 to 100,000 while generating time series or applying deforestation to different queries. As expected, the speedup of deforestation increases as the number of agents increases for all queries. For 100,000 agents, applying deforestation to each query leads to 3.8×, 5×, and 3.6× speedup respectively, compared to the base case of generating time series.

Hierarchical communication. In hierarchical communication, messages between agents in the same component arrive in fewer rounds than those in different components. To evaluate whether our system exploits this communication pattern effectively, we configure agents to be K-available to agents in other components every K rounds (see Section 3) and 1-available to those in the same component in each round, and increase K from 1 to 20.[10]

Figure 7d shows that the performance increases by 2× for all workloads, due to improved communication and computation locality. While the performance of most workloads improves as K increases, in the population dynamics, $K = 10$ has a slightly better performance than $K = 20$. Because agents send messages to each neighbor in every round, more messages are buffered as K increases. For $K = 20$, twice as many messages need to be delivered than for $K = 10$ while merging, thus taking a longer time. If the time saved from delivering local messages faster (in terms of wall clock time) outweighs the possible increase in the time to merge, then exploiting local communication improves performance.

8 RELATED WORK

We have pointed out in Section 1 that agent-based simulations play an increasingly important role in various fields of research and applications. While many popular agent-based frameworks such as NetLogo [65] and Repast Simphony [51] are available, these are primarily designed as desktop applications and can not efficiently scale as the number of agents increases. As a result, there has been a growing need for distributed agent-based simulation frameworks.

Repast HPC [18] was designed to enhance the scalability of Repast Simphony. Experiments have shown that Repast HPC outperforms other general-purpose distributed simulation frameworks by adopting optimizations such as point-to-point communication and load balancing [18, 32, 34, 48]. However, Repast HPC faces several scalability bottlenecks. For starters, its computational model is based on executing events stored in a global schedule queue [62]. Unlike CloudCity, Repast HPC lacks the concept of "agent programs" that can isolate local agent behaviors from the rest of the simulation logic. Parallelization is achieved by a distributed implementation of the global schedule queue [18, 62], where each node in a cluster maintains a local copy of a subset of the queue. This is less efficient than running agents concurrently in an embarrassingly parallel fashion and can cause high synchronization overhead while maintaining multiple copies of the schedule queue in sync. In addition, communication in Repast HPC requires sending serialized agent objects in "agent

[10]In the economics example, the market agent now updates the value of the stock based on received trader actions, instead of waiting for actions from all traders.

packages” [61], which transfers unnecessary data and causes high communication overhead. It can be observed that existing agent-based simulation frameworks so far have not profited from top-tier systems and data management research.

CloudCity combines both the flexibility of agent-based frameworks and the scalability of BSP-like distributed systems. In BSP, distributed computation is separated into smaller parts that can be executed independently in parallel and later combined. There is no global data shared by distributed machines, allowing more efficient parallelization. Our computational model *weighted hierarchical BSP* improves over the vanilla BSP model by exploiting hierarchical communication patterns in agent-based simulations. This model is a variant of *hierarchical BSP* models, which are adopted to improve the algorithmic analysis of parallel algorithms, typically divide-and-conquer, over BSP [7, 9, 13, 57]. Traditionally, hierarchical BSP models assume that messages arrive immediately in the next round. Our model considers a delay of K rounds for message arrival. Partitions are created based on the message delay as an optimization, with the purpose of increasing data locality and reducing synchronization costs.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we have made a first foray into defining a semantics, system architecture, and optimizations for agent-based simulation engines that is based on current research on scalable data-parallel systems. As our experiments show, while BSP-like systems are widely used in distributed applications, their performances vary drastically depending on the applications and system designs. For instance, Spark and GraphX both support BSP-like computations, but for stateful, message-intensive applications like simulations, expressing the computation as transformations and actions over RDDs – collections of immutable objects – can cause excessive object creation and copying that leads to orders of magnitude slow down compared with BSP systems that support in-place updates, such as Giraph and CloudCity. In contrast, Giraph is designed for stateful computations in a way that makes it quite suitable for implementing agent-based simulations of limited (message) complexity, scaling well as the total number of agents increases. However, providing a full-support for typical agent-based simulations necessitates system changes to the current design. For example, Giraph lacks native support for intermittent computation or polymorphic agents and messages, which requires inefficient workarounds with noticeable performance penalties. We have also shown that our partition optimizations can yield respectable optimization benefits.

So far we have only scratched the surface of the optimizations made possible by hierarchically partitioning simulations and aligning them to the existing hardware and memory hierarchy. By using more aggressive compilation techniques for the fusion of agent threads, and further work along the lines of out-of-order state-update techniques such as our generalized double-buffering approach, we expect the partition-based simulation model to yield considerably higher speedups.

Simulations play an increasingly important role in data science and are fertile ground for further data management research. While it was only within the scope of this paper to hint at this fact, we believe that there are a number of very promising directions for further work, including advanced query languages for analytics on top of simulation outcomes, fusing simulations with databases, and viewing simulations as learning models.

REFERENCES

[1] David Adam. 2020. Special report: The simulations driving the world's response to COVID-19. https://www.nature.com/articles/d41586-020-01003-6

[2] Akka Developers. 2011. Akka Documentation. https://akka.io/

[3] Apache Giraph Developers. 2011. Apache Giraph. https://giraph.apache.org/

[4] Apache Spark Developers. 2018. Apache Spark. https://spark.apache.org

[5] Kiyoshi Arai, Hiroshi Deguchi, and Hiroyuki Matsui. 2006. *Agent-based modeling meets gaming simulation.* Vol. 2. Springer Science & Business Media, Kisarazu, Chiba, Japan.

[6] Steven C. Bankes. 2002. Agent-based modeling: A revolution? *Proceedings of the National Academy of Sciences* 99, suppl 3 (2002), 7199–7200. https://doi.org/10.1073/pnas.072081299 arXiv:https://www.pnas.org/content/99/suppl_3/7199.full.pdf

[7] Martin Beran. 1999. Decomposable Bulk Synchronous Parallel Computers. In *SOFSEM '99, Theory and Practice of Informatics, 26th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 27 - December 4, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1725)*, Jan Pavelka, Gerard Tel, and Miroslav Bartosek (Eds.). Springer, Milovy, Czech Republic, 349–359. https://doi.org/10.1007/3-540-47849-3_22

[8] Béla Bollobás. 2001. *Random Graphs* (2 ed.). Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9780511814068

[9] Olaf Bonorden, Ben H. H. Juurlink, Ingo von Otte, and Ingo Rieping. 1999. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12-16 April 1999, San Juan, Puerto Rico, Proceedings.* IEEE Computer Society, San Juan, Puerto Rico, 99–104. https://doi.org/10.1109/IPPS.1999.760442

[10] Mark Buchanan. 2009. Economics: Meltdown modelling. *Nature* 460, 7256 (2009), 680–682.

[11] Maurício Cantor, Lauren G. Shoemaker, Reniel B. Cabral, César O. Flores, Melinda Varga, and Hal Whitehead. 2015. Multilevel animal societies can emerge from cultural transmission. *Nature Communications* 6, 1 (2015), 8091.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

[13] Hojung Cha and Dongho Lee. 2001. H-BSP: A Hierarchical BSP Computation Model. *J. Supercomput.* 18, 2 (2001), 179–200. https://doi.org/10.1023/A:1008113017444

[14] Amit K. Chattopadhyay, T. Krishna Kumar, and Iain Rice. 2020. A social engineering model for poverty alleviation. *Nature Communications* 11, 1 (2020), 6345.

[15] Jun-Jie Chen, Lei Tan, and Bo Zheng. 2015. Agent-based model with multi-level herding for complex financial systems. *Scientific Reports* 5, 1 (2015), 8399. https://doi.org/10.1038/srep08399

[16] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (8 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[17] Simon Coakley, Marian Gheorghe, Mike Holcombe, Lee Shawn Chin, David Worth, and Chris Greenough. 2012. Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework. In *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems*, Geyong Min, Jia Hu, Lei (Chris) Liu, Laurence Tianruo Yang, Seetharami Seelam, and Laurent Lefèvre (Eds.). IEEE Computer Society, USA, 538–545. https://doi.org/10.1109/HPCC.2012.79

[18] Nicholson T. Collier and Michael J. North. 2013. Parallel agent-based simulation with Repast for High Performance Computing. *Simul.* 89, 10 (2013), 1215–1235. https://doi.org/10.1177/0037549712462620

[19] Nicholson T. Collier, Jonathan Ozik, and Eric R. Tatara. 2020. Experiences in Developing a Distributed Agent-based Modeling Toolkit with Python. In *9th IEEE/ACM Workshop on Python for High-Performance and Scientific Computing, PyHPC@SC 2020, Atlanta, GA, USA, November 13, 2020.* IEEE, Atlanta, GA, USA, 1–12. https://doi.org/10.1109/PyHPC51966.2020.00006

[20] Célian Colon, Stéphane Hallegatte, and Julie Rozenberg. 2021. Criticality analysis of a country's transport network via an agent-based supply chain model. *Nature Sustainability* 4, 3 (2021), 209–215. https://doi.org/10.1038/s41893-020-00649-4

[21] Benjamin D. Dalziel, Mark Novak, James R. Watson, and Stephen P. Ellner. 2021. Collective behaviour can stabilize ecosystems. *Nature Ecology & Evolution* 5, 10 (2021), 1435–1440. https://doi.org/10.1038/s41559-021-01517-w

[22] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2 (2009), 6:1–6:31. https://doi.org/10.1145/1462166.1462167

[23] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, San Francisco, California, USA, 137–150. http://www.usenix.org/

events/osdi04/tech/dean.html

[24] Riccardo Di Clemente and Luciano Pietronero. 2012. Statistical Agent Based Modelization of the Phenomenon of Drug Abuse. *Scientific Reports* 2, 1 (2012), 532.

[25] Yue Dou, Peter Deadman, Marta Berbés-Blázquez, Nathan Vogt, and Oriana Almeida. 2020. Pathways out of poverty through the lens of development resilience: an agent-based simulation. *Ecology and Society* 25, 4 (2020), 28–42.

[26] Charles Efferson, Sonja Vogt, and Ernst Fehr. 2020. The promise and the peril of using social influence to reverse harmful traditions. *Nature Human Behaviour* 4, 1 (2020), 55–68.

[27] Joshua M. Epstein and Robert L. Axtell. 1996. *Growing Artificial Societies: Social Science from Bottom Up.* The MIT Press, USA.

[28] Paul Erdős and Alfréd Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci* 5, 1 (1960), 17–60.

[29] J. Doyne Farmer and Duncan Foley. 2009. The economy needs agent-based modelling. *Nature* 460, 7256 (2009), 685–686. https://doi.org/10.1038/460685a

[30] Neil M Ferguson, Derek AT Cummings, Simon Cauchemez, Christophe Fraser, Steven Riley, Aronrag Meeyai, Sopon Iamsirithaworn, and Donald S Burke. 2005. Strategies for containing an emerging influenza pandemic in Southeast Asia. *Nature* 437, 7056 (2005), 209–214.

[31] Neil M Ferguson, Derek AT Cummings, Christophe Fraser, James C Cajka, Philip C Cooley, and Donald S Burke. 2006. Strategies for mitigating an influenza pandemic. *Nature* 442, 7101 (2006), 448–452.

[32] FLAME developers. 2012. FLAME Overview. http://flame.ac.uk/docs/overview.html Accessed: 2021-02-09.

[33] Flink Gelly Tutorial. 2015. Flink Gelly. https://flink.apache.org/2015/08/24/introducing-gelly-graph-processing-with-apache-flink/#what-is-gelly

[34] Munehiro Fukuda, Christopher Bowzer, Benjamin Phan, and Kasey Cohen. 2017. Collision-Free Agent Migration in Spatial Simulation. In *Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017, Prague, Czech Republic, September 3-6, 2017 (Annuals of Computer Science and Information Systems, Vol. 13)*, Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki (Eds.). IEEE, rague, Czech Republic, 65–73. https://doi.org/10.15439/2017F172

[35] Martin Gardner. 1970. The fanstastic combinations of John Conway's new solitaire game "life". , 120–123 pages.

[36] Nigel Gilbert and Klaus G Troitzsch. 2005. *Simulation for the Social Scientist.* Open University Press, USA.

[37] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, USA, 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[38] Ferdi L. Hellweger, Erik van Sebille, and Neil D. Fredrick. 2014. Biogeographic patterns in ocean microbes emerge in a neutral agent-based model. *Science* 345, 6202 (2014), 1346–1349. https://doi.org/10.1126/science.1254421 arXiv:https://www.science.org/doi/pdf/10.1126/science.1254421

[39] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. 1983. Stochastic blockmodels: First steps. *Social networks* 5, 2 (1983), 109–137.

[40] Imperial College COVID-19 Response Team. 2020. *Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand.* Technical Report. Imperial College.

[41] Dan Jones. 2015. Conflict resolution: Wars without end. *Nature* 519, 7542 (2015), 148–150.

[42] Stamatis Karnouskos and Thiago Nass de Holanda. 2009. Simulation of a Smart Grid City with Software Agents. In *2009 Third UKSim European Symposium on Computer Modeling and Simulation.* IEEE Computer Society, Athens, Greece, 424–429. https://doi.org/10.1109/EMS.2009.53

[43] Moritz Kersting, Andreas Bossert, Leif Sörensen, Benjamin Wacker, and Jan Chr. Schlüter. 2021. Predicting effectiveness of countermeasures during the COVID-19 outbreak in South Africa using agent-based simulation. *Humanities and Social Sciences Communications* 8, 1 (2021), 174. https://doi.org/10.1057/s41599-021-00830-w

[44] May Lim, Richard Metzler, and Yaneer Bar-Yam. 2007. Global Pattern Formation and Ethnic/-Cultural Violence. *Science* 317, 5844 (2007), 1540–1544. https://doi.org/10.1126/science.1142734 arXiv:https://www.science.org/doi/pdf/10.1126/science.1142734

[45] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, Indianapolis, Indiana, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[46] J. F. Mercure, P. Salas, P. Vercoulen, G. Semieniuk, A. Lam, H. Pollitt, P. B. Holden, N. Vakilifard, U. Chewpreecha, N. R. Edwards, and J. E. Vinuales. 2021. Reframing incentives for climate policy action. *Nature Energy* 6, 12 (2021), 1133–1143. https://doi.org/10.1038/s41560-021-00934-2

[47] A Möhring, G Mack, A Zimmermann, A Ferjani, A Schmidt, and S Mann. 2016. Agent-based modeling on a national scale–Experiences from SWISSland. *Agroscope Science* 30, 2016 (2016), 1–56.

[48] Andreu Moreno, Juan J. Rodríguez, Daniel Beltrán, Anna Sikora, Josep Jorba, and Eduardo César. 2019. Designing a benchmark for the performance evaluation of agent-based simulation applications on HPC. *J. Supercomput.* 75, 3 (2019), 1524–1550. https://doi.org/10.1007/s11227-018-2688-8

[49] Ujjal K. Mukherjee, Subhonmesh Bose, Anton Ivanov, Sebastian Souyris, Sridhar Seshadri, Padmavati Sridhar, Ronald Watkins, and Yuqian Xu. 2021. Evaluation of reopening strategies for educational institutions during COVID-19 through agent based simulation. *Scientific Reports* 11, 1 (2021), 6264. https://doi.org/10.1038/s41598-021-84192-y

[50] Leila Niamir, Gregor Kiesewetter, Fabian Wagner, Wolfgang Schöpp, Tatiana Filatova, Alexey Voinov, and Hans Bressers. 2020. Assessing the macroeconomic impacts of individual behavioral changes on carbon emissions. *Climatic Change* 158, 2 (2020), 141–160.

[51] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark J. Bragen, and Pam Sydelko. 2013. Complex adaptive systems modeling with Repast Simphony. *Complex Adapt. Syst. Model.* 1 (2013), 3. https://doi.org/10.1186/2194-3206-1-3

[52] Kirill Orach, Andreas Duit, and Maja Schlüter. 2020. Sustainable natural resource governance under interest group competition in policy-making. *Nature Human Behaviour* 4, 9 (2020), 898–909. https://doi.org/10.1038/s41562-020-0885-y

[53] Constantin-Valentin Pal, Florin Leon, Marcin Paprzycki, and Maria Ganzha. 2020. A Review of Platforms for the Development of Agent Systems. *CoRR* abs/2007.08961 (2020), 1–40. arXiv:2007.08961 https://arxiv.org/abs/2007.08961

[54] R.G. Palmer, W. Brian Arthur, John H. Holland, Blake LeBaron, and Paul Tayler. 1994. Artificial economic life: a simple model of a stockmarket. *Physica D: Nonlinear Phenomena* 75, 1 (1994), 264–274. https://doi.org/10.1016/0167-2789(94)90287-9

[55] Lionel Parreaux and Amir Shaikhha. 2020. Multi-Stage Programming in the Large with Staged Classes. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Virtual, USA) *(GPCE 2020)*. Association for Computing Machinery, New York, NY, USA, 35–49. https://doi.org/10.1145/3425898.3426961

[56] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Squid: type-safe, hygienic, and reusable quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*, Heather Miller, Philipp Haller, and Ondrej Lhoták (Eds.). ACM, Vancouver, BC, Canada, 56–66. https://doi.org/10.1145/3136000.3136005

[57] Pilar de la Torre and Clyde P. Kruskal. 1996. Submachine Locality in the Bulk Synchronous Setting (Extended Abstract). In *Euro-Par '96 Parallel Processing (Lecture Notes in Computer Science, Vol. 1124)*, Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert (Eds.). Springer, Lyon, France, 352–358. https://doi.org/10.1007/BFb0024723

[58] Marco Raberto, Silvano Cincotti, Sergio M. Focardi, and Michele Marchesi. 2001. Agent-based simulation of a financial market. *Physica A: Statistical Mechanics and its Applications* 299, 1 (2001), 319–327. https://doi.org/10.1016/S0378-4371(01)00312-0 Application of Physics in Economic Modelling.

[59] Varun Rai and Adam Douglas Henry. 2016. Agent-based modelling of consumer energy choices. *Nature Climate Change* 6, 6 (2016), 556–562.

[60] Sara Reardon. 2018. How digital drug users could help to halt the US opioid epidemic. https://www.nature.com/articles/d41586-018-05939-8

[61] Repast HPC developers. 2013. Repast HPC Reference Manual. https://repast.github.io/hpc_tutorial/RepastHPC_Demo_01_Step_07.html

[62] Repast HPC developers. 2023. Repast HPC API. https://repast.github.io/docs/api/hpc/repast_hpc/classrepast_1_1_schedule.html#details Accessed: 2023-03-02.

[63] Thomas C. Schelling. 1971. Dynamic models of segregation. *Journal of Mathematical Sociology* 1, 2 (1971), 143–186. https://doi.org/10.1080/0022250X.1971.9989794

[64] Eric Silverman, Umberto Gostoli, Stefano Picascia, Jonatan Almagor, Marc McCann, Richard Shawm, and Claudio Angione. 2021. Situating agent-based modelling in population health research. *Emerging Themes in Epidemiology* 18, Article 10 (2021), 15 pages.

[65] Seth Tisue and Uri Wilensky. 2004. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, Vol. 21. Citeseer, Boston, MA, 16–21.

[66] James M. Trauer, Michael J. Lydeamore, Gregory W. Dalton, David Pilcher, Michael T. Meehan, Emma S. McBryde, Allen C. Cheng, Brett Sutton, and Romain Ragonnet. 2021. Understanding how Victoria, Australia gained control of its second COVID-19 wave. *Nature Communications* 12, 1 (2021), 6266. https://doi.org/10.1038/s41467-021-26558-4

[67] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111. https://doi.org/10.1145/79173.79181

[68] Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *ESOP '88, 2nd European Symposium on Programming (Lecture Notes in Computer Science, Vol. 300)*, Harald Ganzinger (Ed.). Springer, Nancy, France, 344–358. https://doi.org/10.1007/3-540-19027-9_23

[69] M. Mitchell Waldrop. 2018. What if a nuke goes off in Washington, D.C.? https://www.science.org/content/article/what-if-nuke-goes-washington-dc-simulations-artificial-societies-help-planners-cope

[70] Jing Wu, Rayman Mohamed, and Zheng Wang. 2011. Agent-based simulation of the spatial evolution of the historical population in China. *Journal of Historical Geography* 37, 1 (2011), 12–21. https://doi.org/10.1016/j.jhg.2010.03.006

[71] Jinfeng Wu, Xingang Wang, and Bing Pan. 2019. Agent-based simulations of China inbound tourism network. *Scientific Reports* 9, 1 (2019), 12325.

[72] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, San Jose, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[73] Claire Zoellner, Rachel Jennings, Martin Wiedmann, and Renata Ivanek. 2019. EnABLe: An agent-based model to understand Listeria dynamics in food processing facilities. *Scientific Reports* 9, 1 (2019), 495.

Received October 2022; revised January 2023; accepted February 2023