Thèse n° 8865

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

Présentée le 4 septembre 2023

Faculté informatique et communications
Laboratoire de théorie et applications d'analyse de données
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Zilu TIAN

École
polytechnique
fédérale
de Lausanne

2023

To my family: my parents and sister. I love you.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Christoph Koch. He embodies the epitome of a true scientist: trustworthy, insightful, knowledgeable, and kind, coupled with remarkable talent. He has inspired me to be a better person. Working with Christoph has been an incredible opportunity for which I am extremely grateful.

I also want to thank my collaborators and colleagues: Val Tannen, who has patiently devoted much time and effort in shaping the project during our collaboration; Peter Lindner, our postdoc, who had many technical discussions with me and taught me a lot; Sachin Basil John, who kept me company as we stayed up late and worked through many vacations together, cheering each other up; Catherine Gagliardi, our secretary, who has kindly helped me with many things; Lionel Parreaux, our alumni, who helped me with using Squid.

Outside our lab, special thanks go to my thesis committee, whose valuable feedback has greatly improved the presentation of the thesis: Milos Nikolic, Anne-Marie Kermarrec, Dan Olteanu, and Bryan Ford. I am also deeply grateful to people who have helped me professionally at EPFL: Alexandros Dalis, who supervised my semester projects in my first year; Arash Pourhabibi, Dimitrii Ustiugov, Guohao Dou, Hussein Kassir, Mario Drumond, Mark Sutherland, Panos Sioulas, and Siddharth Gupta, for having technical discussions; and administrative assistants, including Stéphanie Baillargues, Marta Bellone, Annalisa Ginocchio, and Anna Cecelia Chapuis.

My Ph.D. journey would not have been the same without the support of my friends and flatmates: Cixian Shen, who saved me from unconsciousness when I passed out; Hanqi Lu, who generously helped me when I fell sick; Simla Harma, together with Dina Mahmoud, we went on many trips and cooked together; Dina is also credited with proofreading the French translation of the abstract; Teresa Yeo, who is my bouldering buddy.

Last but not least, I want to thank my family: my parents and sister. You have always been there for me unconditionally. So many things have happened during the past few years: a pandemic, a flood, a trade war, an actual war, and waves of layoffs. You have provided a loving sanctuary in this turmoil world. I love you.

Lausanne, 24 May 2023 Z. T.

Abstract

Agent-based simulations have been widely applied in many disciplines, by scientists and engineers alike. Scientists use agent-based simulations to tackle global problems, including alleviating poverty (Dou et al., 2020), reducing violence (Jones, 2015; Lim et al., 2007), and predicting the impact of pandemics (Imperial College COVID-19 Response Team, 2020). In industry, engineers use agent-based simulations to reduce cost and improve efficiency, by creating virtual worlds to model different scenarios and explore various designs with fast feedback at low cost (Fraccascia et al., 2020; Iannino et al., 2020; Zoellner et al., 2019). Agent-based simulations play an increasingly prominent role in modern society.

Despite their significance, agent-based simulations have benefited little from the recent progress in computer science, especially on the fronts of parallel computing and data management. While there has been a growing need to simulate at an ever-increasing scale with finer details, developments on systems that support fast execution of large-scale simulations and efficient integration of simulations with existing data science pipeline operators are dragging behind. This creates new challenges and opportunities for computer scientists.

In this work, we make the first foray into defining a clean semantics that serves as the foundation of agent-based simulations, an abstraction that facilitates users to integrate simulations into data science pipelines, a scalable system architecture with efficient optimizations, and a high-level user-friendly programming model. In particular, 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 also propose to encapsulate simulations via a `Simulate` operator, which enables users to compose and nest simulations just like other data science pipeline operators. In addition, we have designed and developed an open-source distributed system for large-scale agent-based simulations, CloudCity, which implements our semantics to improve the locality of computation, communication, and synchronization in simulations. This system contains efficient optimizations to allow fast execution and efficient query of simulation results. To accommodate users from different backgrounds, we have also developed a user-friendly domain-specific language (DSL) embedded in the programming language Scala, which allows users to write parallel agent programs easily, even with little or no background in distributed computing. We experimentally evaluate the performance of our system on a benchmark suite of agent-based simulations and compare it against existing state-of-the-art BSP-like distributed systems, including Spark, GraphX, Giraph, and Flink Gelly, obtaining insights into the impact of various system design choices and optimization on simulation engine performance.

Key words: agent-based simulations, distributed systems, bulk-synchronous parallel processing, compilation, query languages

Résumé

Les simulations à base d'agents ont été largement appliquées dans de nombreuses disciplines, tant par les scientifiques que par les ingénieurs. Les scientifiques utilisent les simulations à base d'agents pour s'attaquer à des problèmes mondiaux tels que l'eradication de la pauvreté (DOU et al., 2020), la réduction de la violence (JONES, 2015 ; LIM et al., 2007) et la prédiction de l'impact des pandémies (IMPERIAL COLLEGE COVID-19 RESPONSE TEAM, 2020). Dans l'industrie, les ingénieurs utilisent les simulations à base d'agents pour réduire les coûts et améliorer l'efficacité, en créant des mondes virtuels pour modéliser des scénarios différents et pour explorer des conceptions diverses avec un retour rapide à faible coût (FRACCASCIA et al., 2020 ; IANNINO et al., 2020 ; ZOELLNER et al., 2019). Les simulations à base d'agents jouent un rôle de plus en plus éminent dans la société moderne.

Malgré leur importance, les simulations à base d'agents ont peu bénéficié des progrès récents en informatique, en particulier dans les domaines du calcul parallèle et de la gestion des données. Alors que la nécessité de simuler à une échelle de plus en plus grande avec des détails plus fins ne cesse de croître, les développements de systèmes permettant une exécution rapide de simulations à grande échelle et une intégration efficace des simulations avec les opérateurs de pipelines des données existants sont en retard. Cela crée de nouveaux défis et opportunités pour les informaticiens.

L'idée centrale de cette thèse est de définir pour la première fois une sémantique concise et précise qui sert de fondement aux simulations à base d'agents, une abstraction qui facilite l'intégration des simulations dans les pipelines des données, une architecture système évolutive avec des optimisations efficaces et un modèle de programmation facile à utiliser. En particulier, nous généralisons le modèle Bulk-Synchronous Parallel (BSP) afin de le rendre plus adapté aux simulations à base d'agents. De telles simulations présentent fréquemment une structure hiérarchique dans leurs schémas de communication, qui peut être exploitée pour améliorer les performances. Nous autorisons la création de partitions réseau artificielles temporaires au cours desquelles les agents se synchronisent uniquement localement au sein de leur groupe, sans compromettre l'exactitude d'une simulation. Nous proposons également d'encapsuler les simulations via un opérateur `Simulate`, qui permet aux utilisateurs de composer et de hiérarchiser les simulations de la même manière que les autres opérateurs de pipelines des données. De plus, nous avons conçu et développé un système distribué open-source pour les simulations à base d'agents à grande échelle, CloudCity, qui met en œuvre notre sémantique pour améliorer la localité du calcul, de la communication et de la synchronisation dans les simulations. Ce système contient des optimisations efficaces permettant une exécution rapide et une interrogation efficace des résultats des simulations. Pour accueillir des utilisateurs de différents secteurs, nous avons également développé un langage spécifique au domaine (DSL), intégré au langage de programmation Scala, qui permet aux utilisateurs d'écrire facilement des programmes d'agents parallèles, même avec peu ou pas de connaissances en informatique distribuée. Nous évaluons expérimentalement les performances de notre système sur une suite de benchmarks de simulations à base d'agents et le comparons à des systèmes distribués de type BSP existants à la pointe de la technologie, notamment Spark, GraphX, Giraph et Flink Gelly, afin d'obtenir des connaissances sur l'impact des différents choix de conception du système et des optimisations sur les performances du moteur de simulation.

Mots clefs : Simulations à base d'agents, systèmes distribués, modèle Bulk-Synchronous Parallel, compilation, langages de requête.

Contents

List of Figures

List of Tables

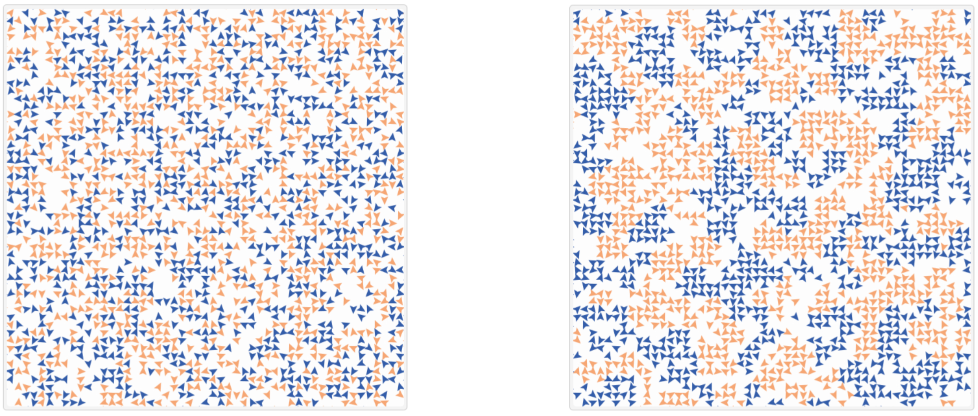
1 Introduction

Agent-based simulations are simulations in which a number of agents, each with their own thread, code, and state, interact in a virtual environment. An agent may execute arbitrary code, affecting its own state as well as the state of the virtual world and any other agents living within. These simulations enable users to make changes to the behavior of one or more individual agents at the micro-scale and observe the collective impact of such changes on the population at the macro-scale.

Developing precise agent-based simulations is relatively simple: The desired individual behavior of an agent can be coded directly, rather than requiring the crafting of complex mathematical models (such as systems of partial differential equations) that describe the collective behavior of all agents indirectly in the aggregate. To see this, consider the classic SIR model that characterizes the spread of an infectious disease in a population as a whole via the following differential equations ("The SIR Model for Spread of Disease - The Differential Equation Model", 2004):

$$\begin{aligned}
\frac{\partial s}{\partial t} &= -\alpha i s, \\
\frac{\partial i}{\partial t} &= 0 - \frac{\partial s}{\partial t} - \frac{\partial r}{\partial t}, \\
\frac{\partial r}{\partial t} &= \beta i,
\end{aligned} \tag{1.1}$$

where s, i, and r are functions of time t that denote the fraction of the population in susceptible (S), infected (I), and recovered (R) states respectively. This model assumes that a person can only be in one of these states, thus the sum of s, i, and r equals one for any time t. When susceptible agents contact infected agents, only a proportion (described by constant α) of these interactions cause susceptible agents to become infected. The infected population recovers at a fixed rate, modeled by another constant β. While these equations can characterize the aggregated behavior of how disease spreads over time, they require non-trivial changes when the complexity of a simulation increases to consider more factors, such as social networks and commute patterns. Agent-based simulations offer a more flexible approach, allowing users to



(a) Initial state (b) After 10 rounds

Figure 1.1: Schelling's Model of Segregation experiment over a 50×50 2D grid using NetLogo (Tisue & Wilensky, 2004). We assume that the population occupies 65% of the cells in the grid. Arrows represent people agents and white cells represent empty locations. The population consists of two equal-numbered races, distinguished by the color of arrows (orange and blue). In each round, every agent checks its neighbors and moves if unhappy. (a) Initially, agents are placed randomly and society is homogeneous. (b) After 10 rounds, society is segregated.

easily adjust agents' states and behavior.

One of the most influential early examples of agent-based simulations is Schelling's Model of Segregation (Schelling, 1969), which demonstrated that discriminatory individual choices can cause then-observed segregation by skin color in the demographic map of the United States. In the original experiment, Schelling abstracted society as a 1D line (Schelling, 1969). He randomly distributed equal numbers of black and white agents, represented by the symbols "+" and "0", along the line. The neighborhood of an agent is defined as extending four agents in both the left and right directions. Schelling assumed that an agent is happy if at least half of its neighbors share the same color as itself. If not, the agent will move to the nearest point in either the left or right direction where it will be happy. Under this assumption, the experiment demonstrated that as agents move, individual preferences cause segregation to occur in the initially homogeneous society. Schelling extended the 1D model to a more complex 2D model in (Schelling, 1971), where society is abstracted like a checkerboard. Every agent occupies a square. The neighborhood consists of eight surrounding squares. The criteria for determining if an agent is happy is identical to that of the 1D experiment. If unhappy, an agent moves to the nearest vacant square that makes it happy. Just like the 1D model, the 2D experiment again demonstrated that agents' individual preferences can lead to segregation in an initially integrated society, as illustrated in Figure 1.1.

Since then, agent-based simulations have become exceedingly complex and played an important role in many fields, including finance and economics (Buchanan, 2009; Chattopadhyay et al., 2020; Chen et al., 2015; Dou et al., 2020; Farmer & Foley, 2009; Raberto et al., 2001; Wu et al., 2019; Zoellner et al., 2019), sustainability research (Chattopadhyay et al., 2020; Colon et al., 2021; Mercure et al., 2021; Möhring et al., 2016; Niamir et al., 2020; Orach et al., 2020; Rai & Henry, 2016), epidemiology (Adam, 2020; Ferguson et al., 2005; Kersting et al., 2021; Mukherjee et al., 2021; Silverman et al., 2021; Trauer et al., 2021), population dynamics (Cantor et al., 2015; Dalziel et al., 2021; Hellweger et al., 2014), and many others (Jones, 2015; Tang et al., 2015; Waldrop, 2018).

For example, a COVID study by epidemiologists at Imperial College in the spring of 2020 (Imperial College COVID-19 Response Team, 2020) garnered worldwide media attention because 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. The prediction results were obtained by adapting a highly complex agent-based simulation model first developed for avian influenza (H5N1) in Southeast Asia (Ferguson et al., 2005). This simulation model used rich information about social networks, commute patterns, occupations, social classes, and the school system on the level of individuals (Ferguson et al., 2005; Imperial College COVID-19 Response Team, 2020). 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 (Ferguson et al., 2005) describes this highly complex data science pipeline that combines a number of simulation models. Agent-based simulations are now crucial tools that help make decisions of monumental importance to modern society.

1.1 Motivation

Practitioners who develop agent-based simulations for their target applications often face a dilemma. On the one hand, the number of available agent-based simulation frameworks appears high on the paper: A recent survey (Pal et al., 2020) listed over 90 frameworks for agent-based simulations, 36 are open-source and general-purpose (as opposed to a specialized framework that is designed for one target application, for example, traffic simulations). On the other hand, the high heterogeneity in the landscape of agent-based simulation frameworks makes it more difficult, as opposed to easier, for users to find the right tool. These simulation frameworks often have inconsistent and incompatible assumptions about agent-based simulations, such as what agents are and how they communicate or synchronize, which drastically increase the learning cost for users.

We illustrate the issue of inconsistent assumptions regarding agent-based simulations in current frameworks by examining three popular general-purpose, agent-based simulation frameworks below, namely NetLogo (Tisue & Wilensky, 2004), Repast family (N. Collier & North, 2011; North et al., 2013), and FLAME (Coakley et al., 2012). For simplicity, we refer to “agent-based simulations” simply as “simulations”.

In NetLogo (Tisue & Wilensky, 2004), a simulation consists of agents in a 2D world, where the world is discretized like a checkerboard. There are four built-in agent types: patches, links, turtles, and the observer: A patch is the smallest square in the 2D world, a link connects two turtles, and turtles move along the patches. The observer instructs what other agents should do in each discrete time tick. The communication model depends on agent types. While the observer can communicate with any other agents, two turtles can communicate only when they have moved to the same patch ("NetLogo Dictionary", n.d.), and two patch agents or link agents cannot communicate directly.

In the Repast family, including Repast Simphony (North et al., 2013) and Repast HPC (N. Collier & North, 2011), users create a simulation by specifying *agent definitions*, a *model*, and at least one *context* and *projection*. Agent definitions are data structures that contain agent states and methods for accessing or updating the states. The model defines the action of agents scheduled at discrete time ticks. Every agent needs to be in a context, a data structure that contains references to agents, which is commonly used to represent a discrete space that agents live in. The relationship between an agent and its context is described by a projection. In Repast Simphony, communication between agents is done by directly modifying the value of other agents' states. Agents can obtain references to other agents through a shared context, which enable them to directly modify the state of others. In RepastHPC, communication between agents requires sending a serialized copy of the sender agent to the receiver. The receiver modifies the state of a deserialized sender agent locally, and (optionally, as specified by users) synchronizes the local copy of the sender agent with other such copies.

In FLAME (Coakley et al., 2012), agents are state-machine-like and users specify agents' states and state transitions in XML. Given agents' current states and available transitions, the FLAME system determines which agents to execute next. Sending a message between two agents is through broadcasting the message to message lists, where each list represents a specific type of messages that an agent can receive according to its state transition, to all machines in a distributed system (FLAME developers, 2021).

This problem of disparate framework-dependent presumptions concerning what agent-based simulations are can be attributed to the lack of a concise and precise abstraction that captures core features of agent-based simulations. Because of this, users can find it difficult to understand what a simulation framework does and how agents behave, which hinders their ability to reason about the correctness of a simulation developed using a third-party simulation framework and their willingness to adopt a framework.

Consequently, users often opt to implement a complex simulation from the ground up, wasting precious time reinventing the wheel to build simulation infrastructure (also referred to as simulation engine) (Cantor et al., 2015; Chattopadhyay et al., 2020; Chen et al., 2015; Colon et al., 2021; Efferson et al., 2020). In the aforementioned COVID study (Imperial College COVID-19 Response Team, 2020), authors also used an in-house agent-based simulation model that was adapted from their previous work (Ferguson et al., 2005), which was built from scratch. Agent-based simulations can greatly benefit from a concise abstraction that captures the core features of agent-based simulations and a general-purpose, scalable simulation engine that can be extended by user-level libraries to be domain-specific.

Agent-based simulations have also become 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. For example, users may be interested in questions such as "How many people are infected after three months, averaged over 5 repeated simulations". Though such use cases are very common, little progress have been made in bridging the semantic gap between the user-level view of agent-based simulations as a tool that constitutes only part of the problem-solving process, and the framework-level view of simulations as standalone programs for large-scale, complex simulations. 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.

Today, there is an increasing need to simulate at an unprecedented large scale, driven by the never-ending pursuit of finer details that expands complex simulations with even more agent behaviors and interactions, but is constrained by the status quo of existing agent-based simulation frameworks. This is a call-to-arm for computer scientists. The main idea of this thesis is to fully unleash the power of agent-based simulations by addressing the observed pitfalls in current simulation frameworks as we have discussed. We aim to provide

- a concise semantics for agent-based simulations that defines what agents are and how agents communicate and synchronize,
- a programming model that allows users to perform complex queries over the output of a simulation, and allows simulations to be integrated into complex data science pipelines, and
- an efficient architecture for general-purpose, agent-based simulation frameworks that can scale to a very large number of agents.

1.2 Challenges

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 in computer science. 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 these 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, "embarassing" 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 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 (Valiant, 1990) (BSP) systems such as Map-Reduce (Dean & Ghemawat, 2004), Spark (Zaharia et al., 2012), and Pregel (Malewicz et al., 2010) 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-

[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.

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 to express 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 to show 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, which allow to obtain better insights and predictions.

1.3 Contributions

We propose a clean semantics for agent-based simulations based on a model of generalized BSP computation that is suitable for processing simulations efficiently, 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 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 synchronizes 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 leveraging (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, which we refer to as weighted hierarchical BSP.

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] In our architecture, a simulation is presented as a data science pipeline operator `Simulate` 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. The optimizations supported by this system include:

- Groups of collocated agents are de-parallelized by turning threads into coroutine objects 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 optimizations that *fuse send and receive operations* to turn message passing into direct memory accesses, and thus remove significant message-passing related compute and communication overheads.

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

- For simulations in which agents are embedded in a graph, we can leverage the spatial attribute of the simulation to reduce the number of agents and messages, by transforming agents into tile agents, which conceptually correspond to a fixed group of vertices in the embedded graph.
- We also explore ways to reduce the semantic gap between application-level user queries and our data model for better performance and improved usability. For instance, we allow users to specify a halting condition via `SimulateUntil`. The user-defined halting condition is evaluated at the end of every round, in the context of the time series obtained so far. A simulation terminates if either the prescribed total number of rounds has passed or the halting condition is evaluated to be true.
- We have also introduced the notion of *virtual rounds*, which allows users to hide certain rounds from advancing agents' clocks, such as those induced by application-level concurrency protocol. This makes it easy for users to calculate aggregated properties of agents without introducing clock agents in their applications.

We introduce a benchmark suite that includes a diverse set of workloads for scenarios such as population dynamics, economics, and epidemiology, which are representative of the types of problems that agent-based simulations are commonly used to address. The goal of the benchmark suite is to provide researchers with a basic skeleton code that can be easily adapted or built upon for their own research purpose. To achieve this, we evaluate in detail how various application-level parameters can be tuned to achieve different outcomes that are of interest to researchers. For our purpose, however, this benchmark suite for agent-based simulations is used to cross-compare the performance of different BSP-like systems when executing representative agent-based simulations.

We compare CloudCity with current state-of-the-art distributed systems that support BSP-like computations, specifically Spark (Zaharia et al., 2012), GraphX (Gonzalez et al., 2014), Flink Gelly (Carbone et al., 2015; Developers, n.d.), and Giraph (Apache Giraph Developers, 2011). Since such systems have different programming models, we explain important details concerning how we implemented the benchmark suite 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 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. Fusing send and receive operations is up to 2× faster than messaging, though using a low-level language with fine-grained memory management, like C/C++, can lead to a higher speedup (6×). We evaluate the effectiveness of tiling using population dynamics. In this example, tiling leads to up to 20× performance improvement for one million agents. To evaluate `SimulateUntil`, we consider a typical user query for an epidemics simulation, "When does at least half of the population become exposed to the disease?", where the population is generated from a random graph. For 100,000 people, `SimulateUntil` is up to 300× faster than `Simulate`. 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.

In a nutshell, the contribution of this thesis can be summarized below:

- a precise and concise theoretical foundation that captures what we believe is the core of agent-based simulations. Our computational model generalizes the well-established bulk-synchronous parallel (BSP) (Valiant, 1990) model, which is widely adopted in popular distributed frameworks in computer science (Apache Giraph Developers, 2011; Dean & Ghemawat, 2004; Malewicz et al., 2010; Zaharia et al., 2012), for efficient processing of agent-based simulations;

- a rich programming model that allows simulations to be integrated into complex data science pipelines as an ordinary analytical operator. Users can express questions about the result of a simulation as queries in a collection-based query language. People who are familiar with object-oriented programming but are not experts in parallel programming can still define parallel agent programs that run in distributed systems at ease using our programming model;

- a scalable system architecture for large-scale agent-based simulation engines, which can serve as a guide for practitioners who develop simulation frameworks; and
- an efficient, open-source, general-purpose, distributed agent-based simulation framework that implements our proposed semantics, programming model, and system architecture, together with efficient system optimizations. We evaluate the performance of our system by comparing it with state-of-the-art open-source BSP-like systems, including Spark, GraphX, Flink Gelly, and Giraph. Our experimental data show that our system achieves on-par or better performance than these systems when evaluated using a comprehensive benchmark suite for agent-based simulations.

Thesis Outline

The content of this thesis expands on our paper (Tian et al., 2023). In Chapter 2, we introduce our programming model, which showcases how users can interact with our distributed simulation engine through a self-contained example. In Chapter 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, illustrated in Section 4. In Chapters 5 and 6, we provide details on translating DSL in the programming model to Scala and exploring system optimizations, respectively. We then describe our benchmark suite and compare our system with other state-of-the-art BSP systems, analyzing both high-level insights on designing scalable systems for agent-based simulations and empirical results that evaluate the performance of our system and the effectiveness of system optimizations in Chapter 7. In Chapter 8, we describe existing distributed agent-based simulation frameworks and their scalability bottlenecks. Finally, we summarize key insights and future work in Chapter 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) embedded in the programming language Scala. A simulation in our system is a round-based concurrent execution of single-threaded agents that communicate by sending messages. For illustration, in this section we regard a simulation as proceeding in a sequence of rounds separated by global synchronizations.[1]

Our programming model makes queries over the outcome of a simulation easy by introducing a data model for simulations and integrating simulations into data science pipelines, which allow users to express an end-to-end decision process that involves agent-based simulations (also referred to as *simulation pipelines*). Users can query and transform the data model of a simulation just like other collections, such as lists or sequences.

In this chapter, we describe our DSL in detail and show how simulations can be composed with other data science pipeline components through `Simulate`. We demonstrate how users can express an entire problem-solving process as a simulation pipeline via a concrete example, where a simulation pipeline consists of different phases that can be composed arbitrarily, including *preprocessing*, *simulation*, and *postprocessing*.

[1]We will explain how agents can synchronize differently later in Chapter 3.

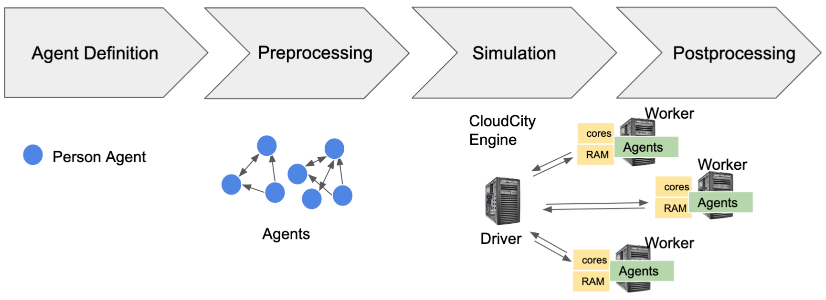


Figure 2.1: A visualization of a sample data science pipeline involving simulations.

2.1 Agent Definition

Agent-based simulation practitioners come from a broad range of fields. We would like users to express parallel agent programs easily using high-level instructions without worrying about low-level details, such as establishing a communication channel between two remote machines prior to sending a message.

To ease the burden of parallel programming, we assume that every agent is single-threaded. Users define an agent type by creating a subclass of `Agent` provided by the CloudCity library. An *agent* is an object instantiated from a subclass of `Agent`.

The assumption that user programs in a distributed framework are single-threaded might appear counter-intuitive, but is not uncommon in practice: Google's MapReduce (Dean & Ghemawat, 2004) and its successors (Apache Hadoop Developers, 2006; Gates et al., 2009; Thusoo et al., 2009; Zaharia et al., 2012) provide a functional interface, where users specify a desired transformation over a parallel collection as a function, in the same way as defining a function over a non-parallel collection in a single-threaded program; Microsoft's Dryad (Isard et al., 2007) structures a distributed application as a dataflow graph and developers provide sequential programs for computational vertices in the graph; NVIDIA's shader language (Mark et al., 2003) allows users to program massively-parallel GPUs using sequential C-like syntax.

Additionally, we assume that agents communicate by sending messages. Each agent is uniquely addressable and has a mailbox. In particular, incoming messages are buffered in the mailbox, waiting passively to be processed by the agent. Below, we describe communication instructions in our DSL. To send a message, an agent uses

```
send(rid:Long,m:Message):Unit,
```

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

```
receive():Option[Message],
```

which returns `None` if the mailbox is empty.

The `send` and `receive` instructions implement a simple message-passing protocol, which is commonly used in distributed applications. In this protocol, each thread has a unique id that is known to other threads in the application. A thread specifies the receiver id when sending a message. This provides a hardware-independent abstraction, that each thread is uniquely addressable regardless of its physical placement, allowing an application developed on a single machine to run directly in a distributed system without any modification to how threads communicate.

By default, we assume a static delay where messages take one round to arrive. The underlying system (our distributed engine) is responsible for delivering messages between agents. Some messages can tolerate a longer delay. For example, an agent may process messages once every 20 rounds. Not all messages 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.

The message-passing protocol places little restriction on what messages contain and how messages are processed. Specifically, a message can be ill-formed and does not contain all arguments that are required for processing. In practice, it is desirable to ensure that messages are well-formed. Hence, we also support remote procedure calls (RPCs) in our DSL, a special type of message-passing protocol that limits senders to only send valid messages that can be processed by receivers. RPCs have two types of messages, requests and replies. In our programming model, the public methods in an agent program are considered RPC methods. An RPC request message contains the identifier of an RPC method defined in the receiver. When processing an RPC request, the receiver looks up the corresponding RPC method and invokes it locally. For performance reasons, we differentiate two types of RPC requests in our DSL, depending on whether a receiver sends an RPC reply message: a *two-sided* RPC request requires the receiver to send an RPC reply message back to the sender, which contains the return value of the local call; a *one-sided* RPC request does not have an associated reply.

Agents send a two-sided RPC request message with

    asyncCall(receiver.API(args*):T,delay:Int):Future[T],

which returns a future object used by the sender to check whether the RPC reply has arrived and to retrieve the return value in the RPC reply. A future object has type `Future[T]`, which is defined in our library, but with a similar usage as that in the standard Scala library. `T` is a type variable that denotes the type of return value in an RPC reply associated with a future object.

Sending a one-sided RPC request can be done using

    callAndForget(receiver.API(args*):T,delay:Int):Unit,

which has similar signature as `asyncCall`, but returns nothing. RPC request messages generated by `asyncCall` and `callAndForget` can be distinguished by a `sessionId` attribute, which is `None` for messages generated for `callAndForget`.

To retrieve an RPC message, agents can use `receive` repeatedly. Alternatively, our DSL allows a receiver to retrieve and process RPC requests in batch via

    handleRPC():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

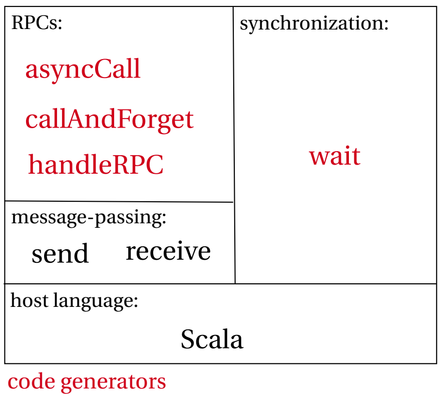


Figure 2.2: DSL stack of CloudCity. Our DSL is embedded in the host language Scala and contains three components: message-passing, RPCs, and synchronization. While message-passing instructions can be implemented as Scala functions directly, instructions in RPCs and synchronization (shown in red) are code generators that require binding-time analysis to dynamically generate corresponding Scala instructions (see Chapter 5).

the same `delay` and `sessionId` as the corresponding request.

We point out that RPC instructions like `asyncCall` and `callAndForget` are higher-order functions that take another function expression `receiver.API(args)` as an input. While this abstraction is convenient for users by isolating them from low-level implementation details such as how to package the corresponding RPC message, this makes it necessary for our system to automatically derive available RPC methods from agent class definitions and assign these methods a unique method identifier before generating the corresponding message. In other words, such instructions are *code generators* that emit different instructions based on static analysis. Similarly, though `handleRPC` is not a higher-order function, it is also a code generator that can produce different instructions for different agent class definitions.

Computations of parallel agents proceed in a sequence of rounds, separated by global synchronization. We introduce a synchronization instruction

```
wait(n:Int):Unit.
```

In round t, an agent that executes `wait(n)` becomes idle until round $t+\mathtt{n}$ starts, and only then it will resume executing instructions that follow `wait(n)`. `wait` is also a code generator. The `wait` instructions split instructions defined in the body of an agent program and generate an array of continuations for the separated instructions as well as the corresponding control logic for executing a continuation when computation resumes.

Overall, our DSL consists of three components: message-passing, RPCs, and synchronization, built on top of the host language Scala, summarized in Figure 2.2. Being an embedded language, our DSL can reuse Scala instructions. We illustrate this in Figure 2.3, which contains the agent definition for a minimal epidemics example, described below.

```
1  @lift
2  class Person extends Agent{
3    var infectious: Boolean = Random.nextBoolean()
4    // include for clarity; outNeighbors is actually defined in Agent
5    // specified during preprocessing phase
6    var outNeighbors: Seq[Person] = Seq[Person]()
7
8    def infect(): Unit = {
9      infectious = true
10   }
11
12   def main(): Unit = {
13     while (true) {
14       handleRPC()
15       if (infectious){
16         outNeighbors.filter(_=>Random.nextBoolean())
17           .foreach(i=>callAndForget(i.infect(), 1))
18       }
19       wait(1)
20     }
21   }
22 }
```

Figure 2.3: Pseudocode of agent definition for Example 2.1.

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 infected indefinitely. The disease spreads as infectious people randomly contact others.*

Agent definition in Figure 2.3 resembles a typical object-oriented program. Each person agent has attributes (lines 3–6) and methods (`infect`, `main`). The `infect` method (lines 8–10) can be specified in RPC requests for execution. On line 17, we use `callAndForget` to send RPC requests for `infect`. A healthy person becomes infectious after processing RPC requests. This agent definition is staged (annotation on line 1) via Squid (Parreaux & Shaikhha, 2020; Parreaux et al., 2017) to let code generators produce instructions in the generated agent program that contains only Scala instructions. Users transform a staged agent program into Scala as shown in Figure 2.4.

The round-based behavior of a person agent is defined in `main` (lines 12–21): in every round, an agent processes all RPC requests using `handleRPC` (line 14). If infectious, an agent infects neighbors randomly selected from `outNeighbors` (lines 15–18). `wait(1)` (line 19) divides the computation of `while(true)` loop (lines 13–20) into different rounds.

Our DSL makes it easy for users to define parallel agent programs using high-level instructions. For example, users do not need to explicitly establish a communication channel between sender and receiver machines before sending a message, nor do they need to write additional instructions for declaring a message type or an RPC service. Even without any background in

```
object Example extends App {
  // use Squid classlifting to transform Person definition into IR
  // IR still contains code generators in CloudCity DSL
  val cls1: ClassWithObject[Person] = Person.reflect(IR)
  // rewrite squid IR into Scala in the generated Person class
  // denote the generated class as generated.Person to distinguish
  compile(cls1)
}
```

Figure 2.4: Pseudocode of transforming the staged `Person` class definition to let code generators in our DSL produce instructions in the generated Scala class definition `generated.Person`

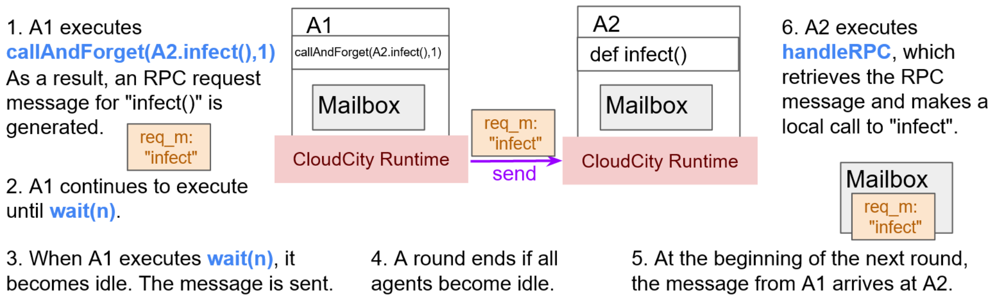

1. A1 executes callAndForget(A2.infect(),1) As a result, an RPC request message for "infect()" is generated.
req_m: "infect"
2. A1 continues to execute until wait(n).
A1
callAndForget(A2.infect(),1)
Mailbox
CloudCity Runtime
req_m: "infect"
send
A2
def infect()
Mailbox
CloudCity Runtime
6. A2 executes handleRPC, which retrieves the RPC message and makes a local call to "infect".
Mailbox
req_m: "infect"
3. When A1 executes wait(n), it becomes idle. The message is sent.
4. A round ends if all agents become idle.
5. At the beginning of the next round, the message from A1 arrives at A2.

Figure 2.5: A step-by-step breakdown of how two agents A_1 and A_2 in Example 2.1 communicate in the view of users. We highlight DSL instructions in blue.

parallel computing, users who are familiar with object-oriented programming can write agent programs that run on a distributed system. Figure 2.5 shows a step-by-step breakdown of how two agents in a distributed simulation communicate using our DSL, in the view of users.[2] Agent A_1 sends a one-sided RPC request to call an RPC method `infect` defined in A_2. The "CloudCity Runtime" (our distributed engine, described later in Chapter 4) shown in Figure 2.5 is responsible for delivering messages between different machines.

2.2 Preprocessing

The preprocessing phase specifies application-dependent parameters for a simulation, such as how agents are connected and the number of different types of agents. For our running example, we assume that the connectivity of agents is modeled using a random graph model, the Erdős-Rényi model (ERM) (Erdős & Rényi, 1960), where each pair of distinct people agents are connected with the same probability p, shown in Figure 2.6. Users can tune the number of people agents and the value of p for the random graph (line 2). In this example, we create agents (line 3) and connect them as specified by the ERM graph (lines 5–6) for preprocessing. The output of this stage is a collection of agents that can be executed by CloudCity.

[2]We simplify details such as how the system knows when all agents become idle, which do not concern users.

```
object preprocess {
  def apply(population: Int, p: Double): Col[Agent] = {
    // generated.person denotes the generated Person class
    val people = (1 to population).map(new generated.Person)
    // ErdosRenyiGraph, connect outNeighbors
    people.map(n => n.outNeighbors =
      people.filter(_ => _!=n && Random.nextDouble()<p))
  }
}
```

Figure 2.6: Psuedocode of preprocessing for Example 2.1.

2.3 Simulate Operator

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

`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 total number of rounds `total` to be executed by the simulation. To support wide-range user queries over the output of a simulation, which may include nesting different simulations, our data model comprises the time series of all agents, i.e., a sequence of snapshots, where each snapshot is a collection of agent objects taken at the end of each round. The agent collection that is part of the input to `Simulate` can come from either preprocessing or the time series generated by other simulations, thus allowing different simulations to be nested and composed.

In large-scale simulations, auxiliary simulations are common for reconstructing missing data or comparing different configurations, as seen in the epidemics study from (Imperial College COVID-19 Response Team, 2020) in Chapter 1. Such simulations form complex data science pipelines and can greatly benefit from composing simulations as an operator with other components in the pipeline, as we will show in the discussion of postprocessing below.

2.4 Postprocessing

In postprocessing, users express analytical questions over the output of a simulation. While different in nature, we draw parallels to typical data management and querying problems: questions about an agent-based simulation can be seen as queries (in the usual sense) over the time series using a collection-based language.

To illustrate, we summarize some typical questions for the epidemic simulation in Example 2.1 and present their corresponding query expressions in Table 2.1. Our questions are representa-

Table 2.1: Sample questions for epidemic simulations and the respective query expressions using the `Simulate` operator.

Q1	How does the total number of infected people change over time? `Simulate(agents,90).map(_.filter(_.asInstanceOf[Person].infectious).size)`
Q2	How many individuals are infected at the end of the simulation, averaged over 10 simulations? `Range(1,10).map(Simulate(agents,90).last.filter(_.asInstanceOf[Person].infectious).size).average`

Q3	How many people are infected at the end of the simulation, if the value of `p` in the random graph model is 0.001, 0.01, or 0.1, respectively?
	`Seq(0.001,0.01,0.1).map(p ⇒ Simulate(preprocess(10000,p),90).last`
	`.filter(_.asInstanceOf[Person].infectious).size)`

Q4	Start a new simulation from day 30 of an existing simulation. What is the infected population at day 30 of the new simulation? `Simulate(Simulate(agents,30).last,30).last.filter(_.asInstanceOf[Person].infectious).size`

tive of a range of analytical questions. Users can calculate a statistical property of all agents in the time series (Q1), aggregate results from multiple simulations (Q2 and Q3), and nest different simulations (Q4). The `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. The number of agents is 10,000.

We now show the pseudocode of an end-to-end example for Example 2.1 in Figure 2.7, using Q1 as a sample user query in postprocessing.

```
object EpidemicSimulation{
  def apply(population: Int, p: Double): Seq[Int] = {
    // Preprocess
    val agents = preprocess(population, p)
    // Start simulation
    val timeseries = Simulate(agents, 90)
    // Postprocess, query evaluation
    timeseries.map(t => t.filter(a => a
      .asInstanceOf[Person].infectious==true).size)
  }
}
```

Figure 2.7: Pseudocode of an end-to-end example for Example 2.1.

3 Model of Computation

As presented so far, we have explained that an agent is single-threaded and communicates with other agents by sending messages. We have also mentioned that an agent-based simulation in our system is a round-based execution of parallel agents, but have yet to justify the motivation behind this design choice or explain the underlying mechanisms for computation, communication, and synchronization. This is the main focus of the current chapter.

3.1 Desiderata

Conceptually, each agent has attributes that correspond to its own characteristics and a mental model of the *world*, which we will elaborate on shortly. These agent attributes are private and can only be accessed by the owner agent. Agents also have behaviors, which are defined by instructions that may read and write their attributes, including communicating with other agents by messaging.

A world is a sandbox in which agents live and can be modeled in distinct ways. On one end of the spectrum, a world is a shared container with mutable states, where any within agent can modify the state of the world and such changes are immediately visible to other agents in the world. In other words, the world is a global variable shared by agents.

The idea of implementing a world as a global variable has been employed in existing agent-based simulation frameworks like Repast Simphony (North et al., 2013), where the “context” of a simulation corresponds to a mutable container that represents the world. While intuitive to understand, the downside of this approach is that *concurrent* modifications to this global variable can cause race conditions and data corruption. However, in Repast Simphony, such concurrency concerns do not arise, because the execution is single-threaded. Users schedule at which time tick should an event be executed, which can modify the global variable. The user-level concurrency model is that multiple events can be scheduled to occur at the same tick. Yet, the scheduled events are placed into an event queue and executed strictly sequentially (North et al., 2013; Repast HPC developers, 2023).

Parallelizing a simulation where a world is a global variable requires substantial changes to both simulation engines and existing user programs, compared with its non-parallelized counterpart. For starters, a world should be implemented using a concurrent data structure that supports parallel operations. User programs need to be adapted to use low-level atomic instructions when describing how agents interact with a world, which is error-prone and drastically increases the learning curve of simulation developers.

On the other end of the spectrum, a world can be viewed as an abstraction for its constituent agents. Put differently, a simulation contains only a collection of communicating agents, who conceptualize the notion of a world in its mental model through interacting with other agents. This simplifies what an agent-based simulation is: An agent-based simulation is a parallel program that consists of single-threaded communicating processes, i.e. agents. This model of parallel programs has been investigated in classic theoretical computer science, encompassing various process calculi such as CCS (Milner, 1980), π-calculus (Milner, 1999), CSP (Hoare, 1978), and ACP (Bergstra & Klop, 1985). Moreover, abstract parallel machines, including bulk-synchronous parallel (BSP) processing model (Valiant, 1990) and its hierarchical variants (Beran, 1999; Cha & Lee, 2001; Torre & Kruskal, 1996), are also designed for this model of parallel programs.

The two approaches to modeling a world can give rise to different design patterns at the user level. To clarify, consider a simple ecology simulation example where the world is a 2D array and each grid/cell contains some amount of grass and may be occupied by a wolf or rabbit. Rabbits eat grass and wolves eat rabbits. Rabbits and wolves can move to different grids.

- In the first approach, users can simulate rabbits and wolves as agents and consider the world as a shared 2D array. Rabbits and wolves directly modify the content of the world. For single-threaded execution, changes to the world are applied consecutively. For parallel execution, however, it is up to users to ensure that concurrent updates to the world are executed as intended, without causing any data corruption or inconsistency;
- In the second approach, there are multiple ways to implement the simulation. One possibility is to introduce a world agent that interacts with rabbit and wolf agents. Rabbits and wolves communicate with the world agent by sending messages. Alternatively, users can simulate each grid in the 2D array as an agent, which may contain a rabbit, wolf, or some grass. Grids communicate to implement the desired simulation behavior. Agents can be easily parallelized.

We point out that in the second approach, while users avoid using low-level error-prone atomic instructions, they still need to implement possible conflict resolution logic in agent programs to handle concurrency, by specifying how an agent processes messages. For instance, if multiple rabbits want to move to the same cell simultaneously, then the world agent will receive one message from each of the rabbits and should process these messages sequentially in a way that satisfies the application requirement.

In general, we do not assume ordering for concurrent messages that arrive in the mailbox of an agent. An agent can process these messages in any order. However, if users want to make sure that a collection of messages are always processed in the same order, then users can program the receiver agent to always sort messages before processing them. Messages can be sorted by the numeric value of sender ids (if there are no concurrent messages from the same sender agent at the receiver) or other desired key values that are encoded in messages.

Another desired feature that is prevalent in agent-based simulations is round-based semantics, like in a board game. Intuitively, the popularity of round-based semantics lies in its straightforward event ordering. An event is a user-level abstraction of a block of instructions that will be executed atomically in a desired round. Events that occur in an earlier round should happen strictly before events in a later round, and events that occur in the same round are considered concurrent[1], therefore can be interleaved arbitrarily. In the round-based semantics, a simulation proceeds in a sequence of rounds. Per round, all events in the current round are executed in parallel. After these events have been completed, the simulation advances to the next round. For example, Repast Simphony is round-based, where users define events and schedule these events at different ticks, each tick corresponding to a round. A simulation proceeds by executing all events scheduled at the current tick, though sequentially in Repast Simphony, and then adding the tick by one.

Unlike asynchronous semantics, where every agent has a logical clock that can diverge arbitrarily, round-based semantics lifts the burden of synchronizing parallel agents from users. An agent in round t knows that all other agents, specifically neighboring agents that it communicates with, are also in the same round. The round-based semantics greatly simplifies the programming model of a system.

In light of our discussions, it is easy to see why we consider an agent-based simulation as a round-based execution of parallel agents, which are single-threaded and communicate by sending messages, without any mutable global variables. The round-based semantics closely resembles BSP, where computation proceeds in a sequence of supersteps. Subsequently, we first make some basic notions clear. After that, we investigate the suitability of BSP and hierarchical BSP as a computational model for agent-based simulations, while making it precise the computation, communication, and synchronization mechanisms of these models.

3.2 Preliminary

Definition 3.1 (Agents)**.** *An agent is single-threaded (sequential) and 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.

[1] Regardless of how these events are physically executed, which can be sequential.

The agents' logical clocks are not general logical clocks but much more well-behaved: They are updated *only through the synchronization mechanism of a system, as described in the corresponding computational model, and can't be otherwise manipulated, in particular not by agents themselves.*

The clock of every agent gets updated in increments of one. Therefore, a *time series* of a 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.

We explain each model in terms of the following agent instructions:

- `send(m,B)`: send message `m` to agent `B`.
- `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.
- `receive`: fetch an arbitrary message from the mailbox (or return nothing if the mailbox is empty).

This matches our description in Chapter 2. We further assume that all messages sent during the simulation are unique.

For a collection of agents, we describe how such agents can be partitioned and merged using *weighted hierarchical partition.* Assume there exists a recursive partition of the agents. Per set C in this recursive partition, we can fix a separate number K (the "weight" of C), indicating that this set of agents is synchronized every K rounds. We define the weighted hierarchical partition more precisely below.

Definition 3.2 (Weighted hierarchical partition)**.** *Let $S = \{\mathtt{A}_1, \ldots, \mathtt{A}_n\}$ be the set of all agents. A weighted hierarchical partition of S is a tree $\mathscr{C} = (V, E)$ with two vertex-labeling functions $C\colon V \to 2^S$ and $K\colon V \to \mathbb{N}$ such that*

- *for the root vertex r we have $C(r) = S$;*
- *for any non-leaf vertex v of $\mathscr{C}$ 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 $\mathscr{C}$ such that $C = C(v)$ the *components* of $\mathscr{C}$. 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.

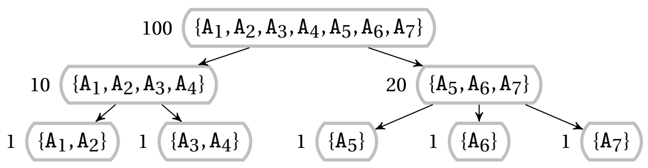


Figure 3.1: A hierarchical partition of a set of 7 agents.

Example 3.3. *Figure 3.1 depicts an example hierarchical partition of the agents* $\mathtt{A}_1, \ldots, \mathtt{A}_7$*. The labels indicate the parameters* $K(C)$*.*

By definition, it is easy to see that a weighted hierarchical partition can be used to describe statically how to partition and merge agents for the entire duration of a simulation. Informally, agents in each component C are synchronized every $K(C)$ rounds. We explain this in more detail when describing partitioned execution later in Section 3.4.

Messages between agents in different components are delivered only when synchronization occurs. Hence, for an arbitrary algorithm where some messages need to arrive in a predetermined number of rounds, a weighted hierarchical partition can violate the message latency constraint. For example, for a message `m` that needs to arrive in 5 rounds, if its sender and receiver components synchronize every 10 rounds in a weighted hierarchical partition, then the latency constraint cannot be satisfied. In other words, for a given algorithm, some weighted hierarchical partitions can be invalid. We capture this by defining the *condition for valid weighted hierarchical partition* below.

For ease of presentation, we make message delay explicit in `send(m,B,k)`. We say `m` is *expected to arrive* at time $t+\mathtt{k}$, where t is the time of `A` at the point of sending. Agent `B` can process a message `m` only if its clock has reached the expected arrival time of `m`.

Condition 3.4. *With fixing* $\mathscr{C}$ *we impose a restriction on the usage of* `send` *instructions in order to match the intuitive meaning of* $\mathscr{C}$ *we described before: If agent* `A` *executes* `send(m,B,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* `A` *and* `B`*.*

If an agent `A` in a simulation attempts to send a message violating Condition 3.4, an exception is thrown, and the simulation is aborted. When this occurs, we say partition $\mathscr{C}$ (in Condition 3.4) is not valid for the simulation. Otherwise, $\mathscr{C}$ is *valid.*

Agents that are separated into different components can only communicate locally with other agents in the same component, until components merge. In other words, the *availability* of one agent to those in other components, where a request message should be delivered and possibly result in a response in a distributed system (Gilbert & Lynch, 2002),[2] is impacted by the partition. Messages sent between agents in different components are delivered only when

[2] In our case, we only consider one-sided messages that do not generate responses.

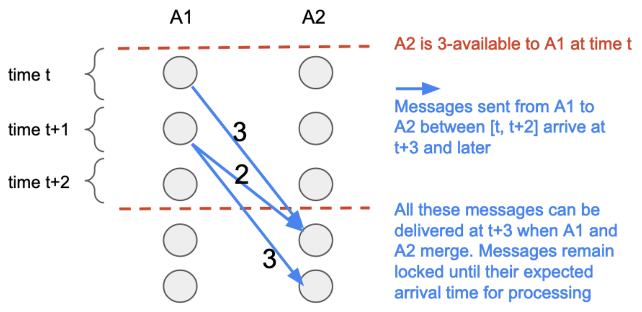

A1
A2
time t
time t+1
time t+2
A2 is 3-available to A1 at time t
Messages sent from A1 to A2 between [t, t+2] arrive at t+3 and later
All these messages can be delivered at t+3 when A1 and A2 merge. Messages remain locked until their expected arrival time for processing
3
2
3

Figure 3.2: An example of K-availability ($\mathscr{A}_t(\mathtt{A}_2, \mathtt{A}_1, 3)$)

merging, which occurs after K rounds. We describe this property as K-*availability*, shown in Figure 3.2.

Definition 3.5 (K-availability)**.** *We discuss the K-availability of agents only in the context of BSP-like models, where there exists a mechanism to synchronize* A *and* B *and ensure that their clocks always have the same value. Let* A *and* B *be two agents. In round t, we say that* A *is* K-available[3] *to* B*, if messages from* B *to* A *sent in the interval $[t, t+K-1]$, are expected to arrive*[4] *at* A *at time $t+K$ or later, represented as $\mathscr{A}_t(\mathtt{A}, \mathtt{B}, K)$. It follows that if* A *is K-available to* B *in round t for some $K > 1$, then necessarily,* A *is $(K-1)$-available to* B *in round $t+1$.*

While weighted hierarchical partitioning can be viewed as a *global configuration*, K-availability is an *individual property* of agents. We examine the relationship between them, such as how to obtain one from another and vice versa. We start with determining K-availability based on a weighted hierarchical partition.

Determine K-availability based on weighted hierarchical partition. We can obtain the K-availability of an agent (referred to as "target agent" in our discussion below) to other agents by walking down the path from the root in $\mathscr{C}$ to the leaf component that contains the target agent. In each component C that contains the target agent, the availability of the target agent to other agents in the same component is updated to $K(C)$, shown in Figure 3.3. To illustrate, Table 3.1 demonstrates how to compute the availability of $\mathtt{A}_1$ in Figure 3.1 using this algorithm. This availability result applies to every K round. In other words, the target agent is K-available to another agent in round αK, where $\alpha \in \mathbb{N}$.

We point out that our weighted hierarchical partition is a theoretical model that describes partitioning abstractly for ease of presentation. `Component` shown in Figure 3.3 closely matches the description of the weighted hierarchical partition, also for illustration; in practice, storing agents repeatedly in each level of the tree is inefficient.

[3]This is different from K-safety, where a system has K replicas of data.

[4]For messages with fixed message delay, this refers to the expected arrival time of a message. For messages that take an arbitrary number of rounds to arrive, their expected time to arrive is conceptually infinite, hence any $K \geq 1$ would satisfy.

```
// A weighted hierarchical partition is a tree
class Component(
  val agents: List[Agent],    // agents in a component
  val K: Int,                 // K value
  val children: Seq[Component]  // Empty if leaf
)

def kavailability(c: Component, target: Agent): Map[Agent, Int] = {
  // store the target agent's availability to other agents
  val result: Map[Agent, Int] = Map[Agent, Int]()

  // walk the tree
  while (c.agents.contains(target)) {
    // update "result" with the component's K value
    c.agents.foreach(a => result(a) = c.K)

    if (!c.children.isEmpty) {
      c = c.children.filter(i => i.agents.contains(target)).head
    } else {
      // return target's kavailability for other agents
      return result.filterKeys(_ != target)
    }
  }
  result.filterKeys(_ != target)
}
```

Figure 3.3: Pseudocode for calculating the K-availability of a target agent to other agents, based on a weighted hierarchical partition.

	A_2	A_3	A_4	A_5	A_6	A_7	Component
A_1	100	100	100	100	100	100	$\{A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$
A_1	10	10	10	100	100	100	$\{A_1, A_2, A_3, A_4\}$
A_1	1	10	10	100	100	100	$\{A_1, A_2\}$

Table 3.1: Compute A_1's K-availability with respect to other agents for Figure 3.1, by traversing from the root component to the leaf that contains A_1. Each row corresponds to one traversal and the last row is final. In each component C, the availability of the target agent to others in the same component is updated to $K(C)$. Each cell in a row contains the updated value for A_1's K-availability to the agent in the corresponding column after visiting a component (shown in the last column).

$$\begin{tikzcd}
0 \arrow[r, "\mathscr{A}_0(\mathtt{A},\mathtt{B},K_0)"'] & K_0 \arrow[r, "\mathscr{A}_{K_0}(\mathtt{A},\mathtt{B},K_1)"'] & K_0+K_1 \arrow[r, dotted, no head] & \sum_{n=0}^{i} K_n \arrow[r, "\mathscr{A}_{\sum_{n=0}^{i} K_n}(\mathtt{A},\mathtt{B},K_{i+1})"'] & {} \arrow[r, dotted, no head] & {}
\end{tikzcd}$$

Figure 3.4: Visualization of A's K-availability to B in a simulation, at different time t

$$\begin{tikzcd}
0 \arrow[r, "\mathscr{A}_0(\mathtt{A},\mathtt{B},K)"'] & K \arrow[r, "\mathscr{A}_K(\mathtt{A},\mathtt{B},K)"'] & 2K \arrow[r, dotted, no head] & iK \arrow[r, "\mathscr{A}_{iK}(\mathtt{A},\mathtt{B},K)"'] & {} \arrow[r, dotted, no head] & {}
\end{tikzcd}$$

Figure 3.5: Visualization of A's K-availability to B in a simulation, after applying Equation (3.7). To distinguish, we drop the subscript t and denote such a sequence as $\mathscr{A}(\mathtt{A},\mathtt{B},K)$ to represent that A is K-available to B every K rounds.

Determine weighted hierarchical partition based on K-availability. We now describe how to determine a valid weighted hierarchical partition based on K-availability, which is more subtle because K-availability is defined for any round value t, whereas weighted hierarchical partition describes periodic behavior that repeats every K rounds. To illustrate, consider the availability of an agent A to B in different rounds t described by the following sequence:

$$\mathscr{A}_0(\mathtt{A},\mathtt{B},K_0),\mathscr{A}_{K_0}(\mathtt{A},\mathtt{B},K_1),\ldots,\mathscr{A}_{\sum_{n=0}^{i} K_n}(\mathtt{A},\mathtt{B},K_{i+1}),\ldots. \tag{3.6}$$

In round 0, A is K_0-available to B. By the definition of K-availability, A is necessarily $(K_0 - t)$-available to B in round t, for $0 < t < K_0$; in round K_0, A is K_1-available to B. Consequently, A is $(K_1 - t)$-available to B in round $K_0 + t$, for $0 < t < K_1$; so on and so force. We can visualize this sequence in Figure 3.4.

To determine a weighted hierarchical partition from K-availability, we first need to transform the sequence in Equation (3.6) via the following rewrite rule:

$$\left(\mathscr{A}_{\sum_{n=0}^{i} K_n}(\mathtt{A},\mathtt{B},K_{i+1})\right)_{i\in\mathbb{N}} \to \left(\mathscr{A}_{iK}(\mathtt{A},\mathtt{B},K)\right)_{i\in\mathbb{N}}, \tag{3.7}$$

where K is a common divisor of $K_i\,(i \in \mathbb{N})$ that is greater than 0. After rewriting, A is K-available to B every K rounds. To distinguish, we use $\mathscr{A}(\mathtt{A},\mathtt{B},K)$ to represent such a sequence, dropping the time-dependent subscript t in $\mathscr{A}_t(\mathtt{A},\mathtt{B},K)$ to indicate that A is now K-available to B every K rounds, as shown in Figure 3.5 and Figure 3.6. *In the discussion below, we only consider K-availability of the form $\mathscr{A}(\mathtt{A},\mathtt{B},K)$.*

In the case that B does not send any message to A in a simulation, we assume K is infinite, $\mathscr{A}(\mathtt{A},\mathtt{B},\infty)$. For our purpose, any positive integer is considered a divisor of ∞.

It is worth pointing out that K-availability is an *asymmetric* relation. For instance, after applying Equation (3.7), A can be 10-available to B every 10 rounds, but B may be 15-available to A every 15 rounds, shown in Figure 3.7. This clearly poses a problem when determining how frequently A and B should synchronize.

We address this by synchronizing A and B at an interval K that corresponds to a positive

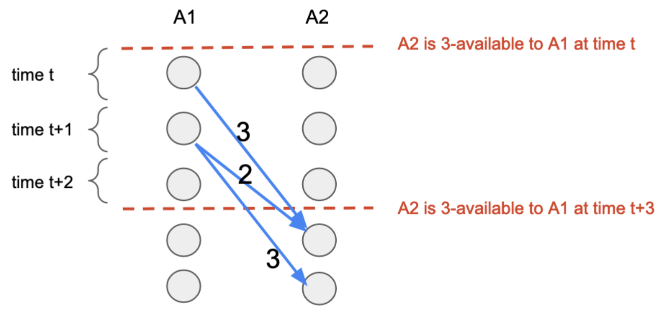

A1
A2
A2 is 3-available to A1 at time t
time t
time t+1
time t+2
3
2
3
A2 is 3-available to A1 at time t+3

Figure 3.6: An example of periodic, time-independent K-availability ($\mathscr{A}(\mathtt{A}_2, \mathtt{A}_1, 3)$, notice that there is no subscript t)

$$\mathscr{A}(\mathtt{A}, \mathtt{B}, K_A):\ 0 \longrightarrow K_A \longrightarrow 2K_A \cdots\cdots iK_A \longrightarrow \cdots\cdots$$

$$\mathscr{A}(\mathtt{B}, \mathtt{A}, K_B):\ 0 \longrightarrow K_B \longrightarrow 2K_B \cdots\cdots iK_B \longrightarrow \cdots\cdots$$

Figure 3.7: Visualization of K-availability being asymmetric. It can happen that A is 10-available to B every 10 rounds ($K_A = 10$), but B is 15-available to A every 15 rounds ($K_B = 15$).

common divisor of their respective K_A and K_B. To distinguish, we denote K-availability relations that are symmetric using $\mathscr{A}^s(\mathtt{A}, \mathtt{B}, K)$, where s stands for *symmetric* (see Figure 3.8):

$$\mathscr{A}(\mathtt{A}, \mathtt{B}, K_A) \wedge \mathscr{A}(\mathtt{B}, \mathtt{A}, K_B) \implies \mathscr{A}^s(\mathtt{A}, \mathtt{B}, K). \tag{3.8}$$

It is easy to see that

$$\mathscr{A}^s(\mathtt{A}, \mathtt{B}, K) \iff \mathscr{A}(\mathtt{A}, \mathtt{B}, K) = \mathscr{A}(\mathtt{B}, \mathtt{A}, K).$$

$\mathscr{A}^s(\mathtt{A}, \mathtt{B}, K)$ corresponds to a *component* in a weighted hierarchical partition, which contains two agents A and B, and the weight of the component is K.

For any simulation, we apply Equation (3.8) to every pair of agents in the simulation, af-

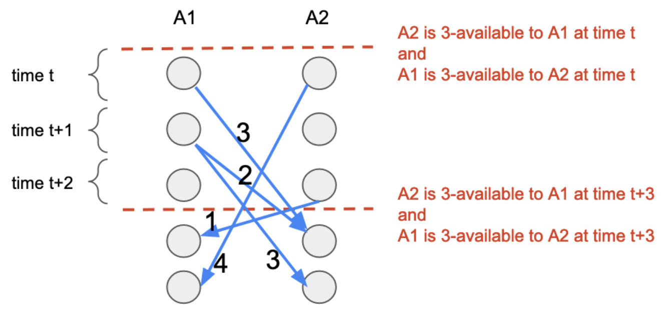

A1
A2
A2 is 3-available to A1 at time t
and
A1 is 3-available to A2 at time t
time t
time t+1
time t+2
3
2
1
4
3
A2 is 3-available to A1 at time t+3
and
A1 is 3-available to A2 at time t+3

Figure 3.8: An example of symmetric, periodic K-availability ($\mathscr{A}^s(\mathtt{A}_2, \mathtt{A}_1, 3)$, notice the superscript s)

$$\mathscr{A}(\mathtt{A},\mathtt{B},K)\colon\ 0 \longrightarrow K \longrightarrow 2K \cdots\cdots\ iK \longrightarrow \cdots\cdots$$

$$\mathscr{A}(\mathtt{B},\mathtt{A},K)\colon\ 0 \longrightarrow K \longrightarrow 2K \cdots\cdots\ iK \longrightarrow \cdots\cdots$$

Figure 3.9: Visualization of $\mathscr{A}^s(A,B,K)$ (the superscript s stands for "symmetric"). After applying the rewrite rule in Equation (3.8), both A and B are now 5-available to each other every 5 rounds (K=5), where 5 is a common divisor of $K_A = 10$ and $K_B = 15$.

ter applying Equation (3.7). For instance, consider a simulation that contains four agents $(\mathtt{A}_1,\mathtt{A}_2,\mathtt{A}_3,\mathtt{A}_4)$, their availability with respect to other agents, after applying the rewrite rules Equation (3.7) and Equation (3.8), can be

$$\mathscr{A}^s(\mathtt{A}_1,\mathtt{A}_2,1),\mathscr{A}^s(\mathtt{A}_3,\mathtt{A}_4,1),\mathscr{A}^s(\mathtt{A}_1,\mathtt{A}_3,10),\mathscr{A}^s(\mathtt{A}_1,\mathtt{A}_4,20),\mathscr{A}^s(\mathtt{A}_2,\mathtt{A}_3,\infty),$$

also shown in Figure 3.10.

We have created a collection of components, each containing exactly two agents with possibly different labels. An agent can be duplicated in these components. Next we need to combine these components to ensure that agents in these components are disjoint.

We first identify *leaf* components, where $K = 1$, as "unvisited" child components. After that, we randomly combine a set of unvisited child components to form a parent component that contains the union of agents in the children. However, not all such combinations of components are valid as a parent component. To be considered as a parent component, the K value of this combined component should satisfy the following conditions, which follow directly from the definition of weighted hierarchical partition:

- K is a multiple of the weight of each child component, and
- K is a common divisor of the set of values K_i that appear in $\mathscr{A}^s(\mathtt{A}_\mathtt{i},\mathtt{A}_\mathtt{j},K_i)$, for all $\mathtt{A}_\mathtt{i}$, $\mathtt{A}_\mathtt{j}$ that are separated in different child components.

If the combined component is a parent, then the child components are marked as "visited". Otherwise, the child components are added back to the collection of unvisited child components. If the parent component contains all agents, then it is the root component and we have built a weighted hierarchical partition. Otherwise, the newly formed parent component is also added to the collection of unvisited child components and can be combined with others.

In Figure 3.10, leaf components are $C(A_1,A_2)$ and $C(A_3,A_4)$. There is only one combination of the leaf components. Combining them results in a parent component $C(A_1,A_2,A_3,A_4)$, and the K value of this parent component is $\mathtt{gcd}(10,20,\infty) = 10$, where gcd stands for the greatest common divisor function. 10 is also a multiple of the weights of child components. This parent component contains all agents and hence is also the root component. Figure 3.11 shows the hierarchical partition tree that is constructed for this minimal example.

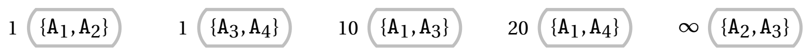


Figure 3.10: An example that contains four agents and their pairwise, symmetric K-availability after applying rewrite rules in Equation (3.7) and Equation (3.8). $\mathscr{A}^s(\texttt{A}_1, \texttt{A}_2, 1)$ corresponds to a component C in weighted hierarchical partitioning that contains agents A_1, A_2 with weight K.

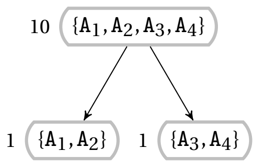


Figure 3.11: A hierarchical partition constructed from components in Figure 3.10.

3.3 BSP Model

The Bulk-Synchronous Parallel (BSP) model is a theoretical model that bridges the programming model of high-level parallel algorithms with that of a parallel machine (Valiant, 1990). A BSP machine assumes several (abstract) cores and a router that delivers messages point to point between pairs of cores. In addition, the BSP machine assumes facilities for periodically synchronizing all cores. The synchronization mechanism can be switched off for a subset of cores (Valiant, 1990).

A main advantage of BSP is its clear and intuitive program semantics. The computation of a parallel algorithm on a BSP machine proceeds in a sequence of *supersteps*. In each superstep, every core executes a task that consists of local computations and communication with other cores by sending and receiving messages. After completing executing a task, cores wait for the current superstep to end before continuing with the next task. A superstep ends if all cores have completed their tasks. Messages arrive at the beginning of a superstep.[5]

BSP has been adopted as the computational model for many state-of-the-art distributed frameworks, including MapReduce (Dean & Ghemawat, 2004, 2010) and Pregel (Malewicz et al., 2010). These frameworks often implement BSP using driver-worker architecture, as illustrated in Figure 3.12: The driver coordinates workers to synchronize, shown as "control flow", and workers can communicate point-to-point, shown as "data flow".

MapReduce. MapReduce was a framework used internally in Google (Dean & Ghemawat, 2004) for distributed data analytics, such as computing word frequency. This framework gave rise to a new programming paradigm under the same name, which inspired many open-source distributed frameworks, including Hadoop (Apache Hadoop Developers, 2006) and Spark (Zaharia et al., 2012), that have now gained wide popularity. In MapReduce, each worker contains a subset of the distributed data. The computation of a program repeatedly proceeds

[5]Though (Valiant, 1990) does not restrict the message latency, i.e. messages can take one or more supersteps to arrive, frameworks including MapReduce and Pregel commonly assume messages have latency one.

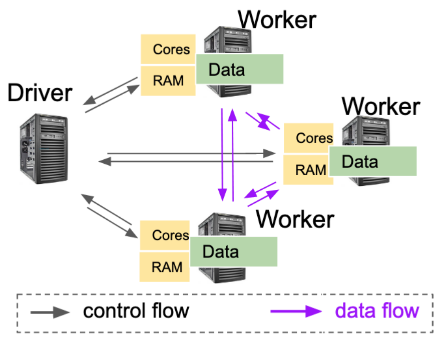


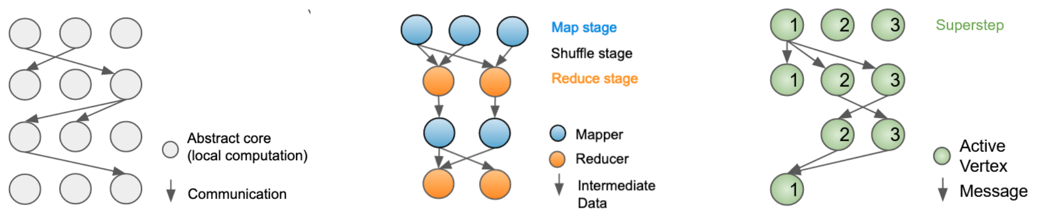
Figure 3.12: Driver-worker architecture: Driver coordinates workers via control messages, shown as control flow; workers are fully connected and can directly send data to other workers, shown as data flow.

in three stages: map, shuffle, and reduce, as illustrated in Figure 3.13b. At the beginning of a map or reduce stage, the driver sends the task to all workers in the next stage. Each worker executes the task over its data in parallel with other workers and notifies the driver when it completes. After all workers in the current stage have completed, the intermediate data generated by the workers are shuffled and delivered before the next stage starts. Each map or reduce stage can be viewed as a BSP superstep.

Pregel. Pregel is designed for large-scale graph processing, such as finding the shortest path between two vertices, in a distributed graph that spans multiple machines. It advocates for "vertex-centric" programming paradigm, in which graph algorithms should be described in terms of how a vertex computes and communicates iteratively, very similar to agent-based programming. The input to Pregel is a graph and a vertex program, which contains a `Compute` method. Each worker contains a portion of the input graph. A vertex is *active* if it has received a message at the beginning of a superstep (except the first superstep, where all vertices are active), and *inactive* otherwise. In each superstep, every active vertex receives messages sent in the previous superstep, executes the `Compute` method, and sends messages, exactly as described in the BSP model. Figure 3.13c demonstrates how executions proceed in Pregel, showing only active vertices.

It is straightforward to see how BSP can be utilized to describe the computation, communication, and synchronization mechanisms for agent-based simulations.

As a model for agent-based simulations (non-partitioned execution). Every agent can be abstracted as a BSP core that can perform arbitrary local computation and communicate through messaging through `send` and `receive` instructions (see Section 3.2). Computation proceeds in "global rounds" where after every round, all agents are synchronized. Each simulation *round* corresponds to a *superstep* in BSP. The `wait` instruction represents that a



(a) BSP (b) MapReduce (c) Pregel

Figure 3.13: Logical views of executions in BSP-like systems (a) an abstract BSP model, where each circle represents a core that performs local computations; (b) MapReduce, where each circle represents a worker in a map or reduce stage; and (c) Pregel, where each circle represents an active vertex labeled by its id. Synchronization occurs at the end of each superstep or stage.

core has completed executing the task of the current superstep. If all cores have completed their tasks, the current superstep ends and the next one begins. Hence, we refer to this model of execution as *non-partitioned execution*, defined below.

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.

3.4 Hierarchical BSP Model

BSP has gained wide popularity in practice due to its simplicity and clarity, making it easy for users to reason about the behavior of their parallel programs. However, a key limitation of BSP is that different subsets of cores in a BSP machine cannot communicate and synchronize locally (Valiant, 1990), which hinders its ability to fully exploit communication locality. The hierarchical BSP model addresses the lack of local communication within subsets of cores in BSP, by allowing a BSP machine to *partition* into multiple BSP machines (Beran, 1999; Cha & Lee, 2001; Torre & Kruskal, 1996), where each new BSP machine contains a disjoint subset of the cores of the parent BSP that now communicate locally, as shown in Figure 3.14. The hierarchical BSP model also allows multiple BSP machines to be *merged* into one BSP machine. After merging, cores that are previously separated on different BSP machines have now become local and can communicate and synchronize.

The hierarchical BSP model has mainly been a topic of interest for theoreticians. For instance, some parallel algorithms, typically divide-and-conquer, have been shown through algorithmic analysis to achieve better performance, when executed on hierarchical BSP instead of BSP (Beran, 1999; Cha & Lee, 2001; Torre & Kruskal, 1996). In practice, however, hierarchical BSP has not been widely adopted by large-scale distributed systems. To the best of our knowledge, Paderborn University BSP library (PUB) (Bonorden et al., 1999) is the only application that

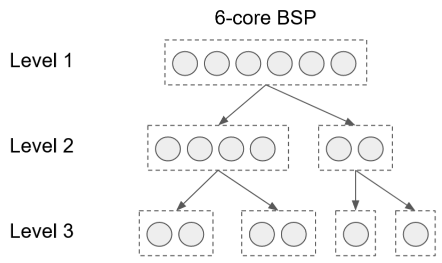


Figure 3.14: A logical view of partitioning in the hierarchical BSP. Initially, there is a single BSP machine that contains 6 cores, shown in level 1. After partitioning, the 6-core BSP machine in level 1 is separated arbitrarily into two BSP machines that contain 4 and 2 cores respectively in level 2. This process can repeat for each BSP machine. The BSP machine with 4 cores in level 2 is partitioned into two machines in level 3, each with 2 cores.

supports hierarchical BSP. We describe PUB briefly below, focusing on how it implements the partition and merge operations.

Paderborn University BSP library (PUB). PUB is a parallel library developed in the programming language C that supports hierarchical BSP (Bonorden et al., 1999). In this library, an abstract BSP machine is implemented as a BSP object that contains fields such as the identifier of the BSP machine, IDs of the first and last cores in the BSP machine, and a pointer to the immediate parent BSP object. New BSP machines are created and removed in a stack-like order (Bonorden et al., 1999).

The syntax for partitioning a BSP machine is (using C syntax, same below)[6]

```
BSP bsp_partition(BSP bsp, BSP subbsp, int n, int[] part),
```

where the variable `bsp` is a pointer to the current BSP machine that will be partitioned; `subbsp` is the return value of this instruction and points to the newly generated BSP machine on top of a BSP stack[7]; `n` denotes the total number of the newly generated BSP machines, and `part` is an array of integers that lists the number of cores that each new BSP machine should contain respectively.

To merge the generated BSP machines, users call

```
void bsp_done(BSP subbsp),
```

which takes a pointer to the generated BSP machines `subbsp` that is returned by `bsp_partition`

[6]We renamed the type name for a BSP object from `tbsp` in (Bonorden et al., 1999) to BSP for clarity.

[7]Though authors of the library mentioned that the BSP machines are managed in a stack-like order, the paper (Bonorden et al., 1999) does not state explicitly about the existence of a “BSP stack”. There is no source code available to check how the stack-like order is implemented. Here we use the term “BSP stack” speculatively, which refers to a data structure that facilitates managing BSP machines in a stack-like order.

as the input and removes these generated BSP machines.

Hierarchical BSP can also be used as a computational model for agent-based simulations. To describe the hierarchical structure of agents created by `partition` and `merge` operations, we use *weighted hierarchical partition* (see Section 3.2).

As a model for agent-based simulations (partitioned execution). Let S be a set of agents and $\mathscr{C}$ be a *weighted hierarchical partition* of S. Synchronization is governed by *local leader* processes for the components, instead of a global leader.[8] Execution under this model is referred to as *partitioned execution*, explained below.

Consider some clock value t. For every agent, there exists some component C in $\mathscr{C}$ 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 as described in the non-partitioned execution (see Section 3.3). In particular, when the leader has received notifications from all agents in C, the clocks of all agents in C get incremented to $t+1$. In general, if C is a component in $\mathscr{C}$, then the agents in C get synchronized every $K(C)$ rounds of their local clocks.

3.5 Partition-Tolerant BSP Model

While hierarchical BSP enables temporary local communication in a BSP machine for better performance, this requires an existing BSP program to be modified with partition and merge operations, as described in the previous section. For instance, PUB implements the hierarchical model by providing user-level instructions that allow users to create partitions or merge explicitly.

We propose *partition-tolerant BSP* model to leverage the communication locality of a BSP program as an optimization. From a high level, our model combines the simple and intuitive semantics of BSP with the efficient execution of hierarchical BSP. When implemented in a system, this model allows users to develop programs at the abstraction level of BSP, but during the execution, BSP cores may get temporarily partitioned and communicate only locally, like hierarchical BSP. Partitioning is artificially introduced in a controlled way, only to enable temporary local communication. In distributed systems, the ability of a system to continue operating despite partial network failure is referred to as partition tolerance (Gilbert & Lynch, 2002). Although the underlying mechanisms that cause partitions are different, our requirement that a BSP program should continue executing even when cores are partitioned, can be viewed as a form of partition tolerance, as reflected in the name of our model.

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

The key property of partition-tolerant BSP is that a BSP program should *behave* the same, both in non-partitioned execution mode (like BSP) and partitioned execution mode (like hierarchical BSP). In the context of agent-based simulations, we characterize the *behavior* of a simulation by the time series generated at the end of each superstep. More concretely, every time series of a simulation under non-partitioned execution should also be a time series under partitioned execution, and vice versa.

The challenge is that partitioning under partitioned execution can cause messages to be delayed and reordered, compared with non-partitioned execution. We examine three distinct cases, each characterized by different assumptions on whether messages have fixed latency. Firstly, messages take an arbitrary number of rounds (more than one) to arrive. Users do not specify when a message is expected to arrive. This makes it impossible for a receiver to distinguish between delays due to partitioning versus those caused by longer transmission times. We refer to this refined model as *asynchronous* partition-tolerant BSP (symbolized by $\mathscr{M}_{\mathsf{async}}$). For any partition configuration, we show that every time series under non-partitioned execution is also a time series under partitioned execution, and in reverse.

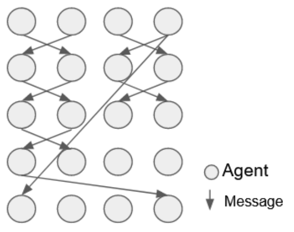
Secondly, messages take a fixed number of rounds to arrive, which may vary among messages. This fine-tuned model is called *synchronous* partition-tolerant BSP (indicated by $\mathscr{M}_{\mathsf{sync}}$). Here, messages cannot be reordered or delayed. We limit ourselves to only *valid* partitions for a simulation (as explained in Section 3.2). For any valid partition, every time series under non-partitioned execution is also a time series under partitioned execution, and conversely.

Finally, some messages, not necessarily all, have a fixed delay, which is termed *partially-synchronous* partition-tolerant BSP (denoted by $\mathscr{M}_{\mathsf{part-sync}}$). Only messages that have fixed delay cannot be reordered or delayed. Similar to $\mathscr{M}_{\mathsf{sync}}$, we only consider *valid* partitions and show that every time series under non-partitioned execution is also a time series under partitioned execution, and the reverse is also true.

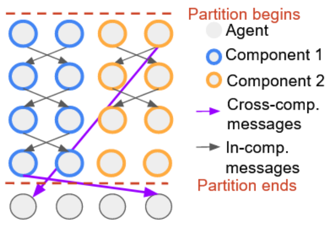
Below we describe $\mathscr{M}_{\mathsf{async}}$, $\mathscr{M}_{\mathsf{sync}}$, and $\mathscr{M}_{\mathsf{part-sync}}$ in detail and discuss the relationship between them. In particular, we show that for a simulation executed under partition-tolerant BSP, a person is unable to distinguish which model ($\mathscr{M}_{\mathsf{async}}$, $\mathscr{M}_{\mathsf{sync}}$, or $\mathscr{M}_{\mathsf{part-sync}}$) this simulation is generated from, based on simulation behavior. In addition, for $\mathscr{M}_{\mathsf{async}}$, we compare it with an asynchronous model that does not assume any synchronization mechanism, and show that a simulation under $\mathscr{M}_{\mathsf{async}}$ can also be obtained from this asynchronous model, and conversely.

Asynchronous Partition-Tolerant BSP ($\mathscr{M}_{\mathsf{async}}$). In $\mathscr{M}_{\mathsf{async}}$, messages take an arbitrary number of rounds to arrive. More concretely, for a message `m` sent using `send(m,B)`, `m` is delivered when `B`'s clock gets incremented to $t+n$, where t is the time of sender `A` at the point of sending and $n \geq 1$. Agent `B` can process a message `m` only if its clock is greater than the sent time of `m`.

Below, we show that every time series of S under non-partitioned execution is a time series under partitioned execution, and vice versa.



(a) Non-partitioned execution



(b) Partitioned execution

Figure 3.15: Logical views of executions in partition-tolerant BSP model: (a) non-partitioned execution, same as the BSP model, and (b) partitioned execution, agents are separated into different components after the partition begins and synchronize locally within each component until the partition ends.

Proof. Let $\mathscr{C}$ be an arbitrary hierarchical partition of a set of agents S. Suppose we have a partitioned execution producing a particular time series. As enforced by the semantics of partitioned execution, 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 remain locked until the clock of the receiver is greater than the sent time of messages. 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 $\mathscr{C}$). Due to this, some messages will reach their target mailbox later. Because messages take an arbitrary number of rounds to arrive, such delay poses no problem. Therefore, this again does not affect the computation or communication, and yields a partitioned execution producing the same time series. $\square$

Synchronous Partition-Tolerant BSP ($\mathscr{M}_{\mathsf{sync}}$). In this model, messages arrive exactly in a prescribed number of rounds, which may vary between different messages, defaulting to one. Conceptually, a set of agents can be partitioned into different subsets in round s temporarily for k rounds, if and only if agents in each subset have agreed in round s to communicate only locally for the next k rounds, as shown in Figure 3.15. In other words, the information that there exists a partition at the beginning of round s, which separates subsets of agents in a way that there will be no communication between any pair of agents in different subsets for the next k rounds, needs to be known before round s starts. In our model, we assume such information is given through a valid partition configuration.

Here we show that in $\mathscr{M}_{\mathsf{sync}}$, every time series of S under non-partitioned execution is a time series under partitioned execution with a valid partition configuration $\mathscr{C}$, and conversely.

Proof. The proof is identical to that of $\mathscr{M}_{\mathsf{async}}$, except for the following. When converting local synchronizations into global ones, messages that arrive earlier in the target mailbox now remain locked until their *scheduled arrival time*. Likewise, when converting global synchronizations into local ones, some messages reach their mailbox later, but such delay poses no problem, as the next synchronization where they can arrive is at most their scheduled arrival time due to Condition 3.4. □

Partially-Synchronous Partition-Tolerant BSP ($\mathscr{M}_{\mathsf{part-sync}}$). Here we consider the case where users specify a fixed delay only for a subset of messages, hence the name *partially-synchronous*. We reserve an invalid message delay value -1 to represent that a message can be arbitrarily delayed. Similar to synchronous partition-tolerant BSP, we assume messages are delivered using `send(m,B,k)`. If $k \neq -1$, message `m` is delivered when `B`'s clock gets incremented to $t + \texttt{k}$, t is the time of sender `A` at the point of sending. Otherwise, `m` is delivered when `B`'s clock gets incremented to $t + \texttt{n}$, $n \geq 1$. The *scheduled arrival time* is defined only for messages sent with $k \neq -1$.

We now show that in $\mathscr{M}_{\mathsf{part-sync}}$, every time series of S under non-partitioned execution is a time series under partitioned execution with a valid partition configuration $\mathscr{C}$, and conversely.

Proof. The proof is identical to that in $\mathscr{M}_{\mathsf{sync}}$, except for the following. When converting local synchronizations into global ones, messages that arrive earlier in the target mailbox remain locked until their scheduled arrival time if their delay is not -1, otherwise, until the receiver's clock is greater than the send time of the message. Similarly, when converting global synchronizations into local ones, some messages reach their mailbox later. If such messages have scheduled arrival time, then Condition 3.4 guarantees that the next synchronization where they can arrive is at most their scheduled arrival time. Otherwise, these messages can be arbitrarily delayed. In both cases, such delay does not cause the behavior of a simulation to change under different execution modes. □

Relationship Between $\mathscr{M}_{\mathsf{async}}$, $\mathscr{M}_{\mathsf{sync}}$, and $\mathscr{M}_{\mathsf{part-sync}}$. In our current setup, the only difference between $\mathscr{M}_{\mathsf{async}}$, $\mathscr{M}_{\mathsf{sync}}$, and $\mathscr{M}_{\mathsf{part-sync}}$ is whether the delay of messages, in the unit of rounds, is prescribed. Intuitively, after a program completes its execution, the delay of every message will be determined, regardless of whether such delays are the same as the expected delay. Here we make this intuition precise, by examining the relationship between $\mathscr{M}_{\mathsf{async}}$, $\mathscr{M}_{\mathsf{sync}}$, and $\mathscr{M}_{\mathsf{part-sync}}$.

For each agent instruction presented in Section 3.2, we introduce a corresponding *event* that is generated during a simulation (equivalently, an execution of a simulation),

- A *send event* $s^t_{\texttt{A}\to\texttt{B}}(\texttt{m})$ occurs when agent `A` issues `send(m,` B`)` in round t.
- A *wait event* $\texttt{wait}^t$ occurs when `A` issues `wait` in round t.

- A *receive event* $r^t_{\mathtt{A} \leftarrow \mathtt{B}}(\mathtt{m})$ occurs when `A` fetches message `m` from its mailbox using `receive` in round t.

An agent executes a sequence of instructions during a simulation, which generates a series of such events. A simulation can be described by a set of events and their relationship.

Depending on the model of partition-tolerant BSP, events are subject to different constraints. To simplify our discussion, we consider a scenario where agents process messages as soon as possible. In other words, a message is fetched using `receive` immediately after the message is unlocked in the mailbox, in the same round.

- In $\mathscr{M}_{\mathsf{async}}$, events are ordered exactly as described in the happened-before relation (Lamport, 1978).
  - Local computation: for any two events a and b in the same agent, a happens before b (denoted as $a \rightarrow b$) if the instruction that generates a comes before that of b in the agent.
  - Message constraint: for any message $\mathtt{m} \in M$, $s^t_{\mathtt{A} \rightarrow \mathtt{B}}(\mathtt{m}) \rightarrow r^{t'}_{\mathtt{A} \leftarrow \mathtt{B}}(\mathtt{m})$, where M denotes the set of all messages in a simulation.
  - Transitive closure: for any event a, b, and c, if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
- In $\mathscr{M}_{\mathsf{sync}}$, events are likewise subject to the happened-before relation. Additionally, *this model is parameterized by a function L that assigns a delay for each message* $\mathtt{m} \in M$ *and introduces a constraint on message latency.*
  - Message latency: for any $\mathtt{m} \in M$ and its corresponding events $s^t_{\mathtt{A} \rightarrow \mathtt{B}}$ and $r^{t'}_{\mathtt{A} \leftarrow \mathtt{B}}(\mathtt{m})$, $t' - t = L(\mathtt{m})$.
- In $\mathscr{M}_{\mathsf{part-sync}}$, event ordering constraints are identical to $\mathscr{M}_{\mathsf{sync}}$, with the only difference that L is defined only for a subset of messages $M' \subseteq M$. The constraint on message latency is updated to the following.
  - Message latency: let $M' \subseteq M$. For any $\mathtt{m} \in M'$ and its corresponding events $s^t_{\mathtt{A} \rightarrow \mathtt{B}}$ and $r^{t'}_{\mathtt{A} \leftarrow \mathtt{B}}(\mathtt{m})$, $t' - t = L(\mathtt{m})$.

We say that a simulation can be obtained under $\mathscr{M}_{\mathsf{async}}$ if its events are ordered by the happened-before relation. In addition to the happened-before relation, if there exists a function $L: M \rightarrow \mathbb{N}$ that assigns a message delay for each message, then this simulation can be obtained under $\mathscr{M}_{\mathsf{sync}}$, with L being a parameter that specifies a fixed latency for each message. Similarly, if events in a simulation are ordered by the happened-before relation and there exists a function $L: M' \rightarrow \mathbb{N}$ defined for a subset of the messages in a simulation, then this simulation can be obtained under $\mathscr{M}_{\mathsf{part-sync}}$, with L being the parameter that prescribes a latency for a subset of messages.

In general, for a simulation obtained under a model $\mathscr{M}_{\mathsf{async}}$, $\mathscr{M}_{\mathsf{sync}}$, or $\mathscr{M}_{\mathsf{part-sync}}$, we show below that a user is unable to tell which model this simulation is obtained from, by demonstrating that it can also be obtained under other models.

Proof. By definition, it is easy to see that for every simulation under $\mathscr{M}_{\mathsf{sync}}$ or $\mathscr{M}_{\mathsf{part-sync}}$, this simulation can also be obtained under $\mathscr{M}_{\mathsf{async}}$.

The converse is also true. We begin by showing that for any simulation under $\mathscr{M}_{\mathsf{async}}$, this simulation can also be obtained under $\mathscr{M}_{\mathsf{sync}}$. More specifically, we need to find a function L that assigns each message an expected latency value consistent with the simulation. This can be achieved by defining L for each message m based on this message's send event $s^{t}_{\mathtt{A}\to\mathtt{B}}$ and receive event $r^{t'}_{\mathtt{A}\leftarrow\mathtt{B}}(\mathtt{m})$ in the simulation, $L(\mathtt{m}) = t' - t$. Hence, this simulation can also be obtained under $\mathscr{M}_{\mathsf{sync}}$ with L being a parameter. The proof that this simulation can also be obtained under $\mathscr{M}_{\mathsf{part-sync}}$ is similar, by constructing L only for a subset of messages. Hence, a simulation under $\mathscr{M}_{\mathsf{async}}$ can also be obtained under $\mathscr{M}_{\mathsf{sync}}$ or $\mathscr{M}_{\mathsf{part-sync}}$, and vice versa.

Similarly, we can show that for any pair-wise combination $(\mathscr{M}_1, \mathscr{M}_2)$ of the models $\mathscr{M}_{\mathsf{async}}$, $\mathscr{M}_{\mathsf{sync}}$, and $\mathscr{M}_{\mathsf{part-sync}}$, for a simulation under $\mathscr{M}_1$, this simulation can also be obtained under $\mathscr{M}_2$, and vice versa. □

$\mathscr{M}_{\mathsf{async}}$ vs Asynchronous. Recall that we say a simulation can be obtained under $\mathscr{M}_{\mathsf{async}}$ if its events are ordered by the happened-before relation. This happened-before relation is also characteristic of asynchronous systems, where messages can arrive at any time (Lamport, 1978). We compare these two models here.

Previously, we use *events* when describing a simulation under $\mathscr{M}_{\mathsf{async}}$. Each event has a timestamp value t that denotes the round value when an event occurs in a simulation. In an asynchronous system, there is no synchronization mechanism that is equivalent to "round". Hence, we need to show that for any simulation under $\mathscr{M}_{\mathsf{async}}$, this simulation can also be obtained under an asynchronous system, that is, with the same timestamp value for each event, and vice versa.

We begin by introducing a timestamp algorithm that updates an agent's clock in an asynchronous system (similar to Lamport's clock (Lamport, 1978)), for each of the agent instructions below.

- `send(m,B)`: send message `m` to agent `B`. The send timestamp of `m` is t, the time of the sender agent at the point of sending.
- `wait`: increment its clock by one and continues working after its clock has been incremented.
- `receive`: fetch an arbitrary message from the mailbox (or return nothing if the mailbox

is empty). If the send timestamp t of the message is greater than or equal to the current time of the receiver, then update the receiver's clock to $t + n$, where $n \geq 1$.

We now revisit the timestamp of events from our previous section. For send and wait events, the timestamp t is the clock value of an agent at the point when a corresponding instruction is executed. For a receive event, however, its timestamp denotes the value of an agent's clock *after* it has been updated.

- A *send event* $s^t_{\texttt{A}\to\texttt{B}}(\texttt{m})$ occurs when agent `A` issues `send(m,` B`)` at time t.
- A *wait event* $\texttt{wait}^t$ occurs when `A` issues `wait` at time t.
- A *receive event* $r^t_{\texttt{A}\leftarrow\texttt{B}}(\texttt{m})$ occurs when `A` fetches message `m` from its mailbox using `receive` and the clock has been updated to time t, which is greater than the send timestamp of `m` by at least one.

By construction, it is easy to see that events in an asynchronous system have the same timestamp value as their $\mathscr{M}_{\mathsf{async}}$ counterpart. Hence, a simulation obtained under $\mathscr{M}_{\mathsf{async}}$ can also be obtained under the asynchronous system, and vice versa.

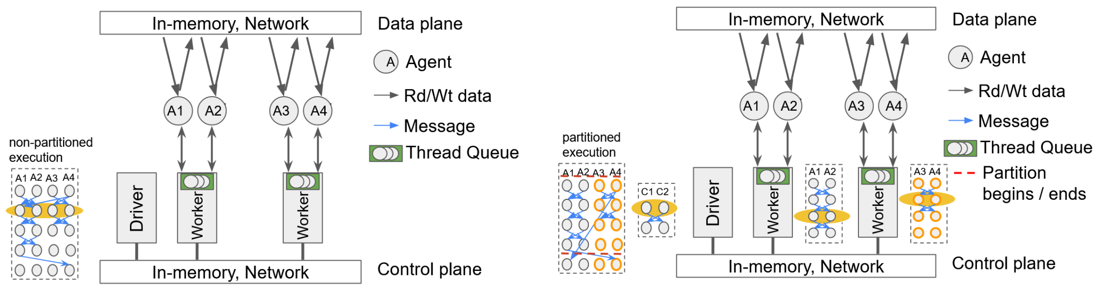
4 System Architecture

The distributed simulation engine in CloudCity is designed to support our partition-tolerant BSP model described in Chapter 3. We have introduced three flavors of partition-tolerant BSP: $\mathscr{M}_{\mathsf{async}}$, $\mathscr{M}_{\mathsf{sync}}$, and $\mathscr{M}_{\mathsf{part-sync}}$. As we have explained, at the system level, the main difference between these models concerns how agents can be partitioned in partitioned execution. For instance, $\mathscr{M}_{\mathsf{sync}}$ and $\mathscr{M}_{\mathsf{part-sync}}$ are defined only for *valid* partitions, that is, messages with a predefined delay can be delivered in such a partition without violating the message latency constrain, whilst $\mathscr{M}_{\mathsf{async}}$ allows agents to be arbitrarily partitioned.

We design our engine in a modular way that can support all three flavors. The difference between the refined models regarding how agents can be re-partitioned is manifested only in the partition algorithm that is used by one driver component, *partition adapter*. Our description of other driver components is independent of, therefore applies equally well to, all refined models of partition-tolerant BSP.

By default, our engine supports $\mathscr{M}_{\mathsf{sync}}$ and $\mathscr{M}_{\mathsf{part-sync}}$ for usability. In many simulations, allowing users to specify when a message should arrive can greatly simplify the design of application logic. We describe how our engine can support $\mathscr{M}_{\mathsf{async}}$ in detail in Appendix B.

The overall structure of CloudCity follows driver-worker architecture. Each worker has a subset of agents, which we refer to as its *local agents*. Workers can communicate directly with each other to deliver messages efficiently. The driver-worker architecture naturally forms a two-level hierarchy. The driver corresponds to the global leader in Chapter 3. For partitioned execution, the driver synchronizes with all workers every $K(S)$ rounds, where S is the set of all agents. For ease of illustration, we assume that the user-defined number of rounds to execute (`total`) is a multiple of $K(S)$. If $K(S)$ exceeds the number of remaining rounds, then the driver synchronizes workers when `total` rounds have been reached. 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 $K(S) = 1$.



(a) Non-partitioned execution (b) Partitioned execution

Figure 4.1: Logical views of our system in two execution modes (a) non-partitioned execution, all agents that are currently executing are in the same round; and (b) partitioned execution, agents in different components can be out of sync while the network partition has not ended. Agents that are currently running on the cores are highlighted with yellow shade.

4.1 Logical View

The logical view abstracts away implementation details of the driver or workers, such as how synchronization is achieved between workers and the driver, and summarizes how our distributed engine implements the computational model from a high level. Figure 4.1 shows our system under the two execution modes of partition-tolerant BSP. The data flow and control flow among the driver and workers are separated into the data plane and control plane respectively. Each worker contains a number of cores and a thread queue. We illustrate agents that are currently utilizing the cores of each worker using a circle labeled with the agent id. The thread queue buffers agents that are waiting to be scheduled in the current round. Messages between agents are delivered either via the network or by passing in-memory pointer references to message objects. Each worker or driver contains a schedule,[1] shown in a dotted square. On the schedule, agents that are currently being executed are highlighted with the shade of brown.

For demonstration, we consider only four agents and two workers in Figure 4.1. Each worker contains two cores and a thread queue. In the non-partitioned execution (shown in Figure 4.1a), all agents in the system proceed in lockstep, synchronizing after every round. Figure 4.1b shows a partitioned execution with two components. Agents A_1 and A_2 are in component 1, and A_3 and A_4 are in component 2. The components are separated into different workers. The partition begins in the first round. Every worker executes independently until the partition ends. During this time, agents in different workers can be out of sync: Agents in worker 1 are in round 3, but agents in worker 2 are in round 2.

The physical view of our distributed engine is shown in Figure 4.2, along with applicable software optimizations. We show the software stack for our system in Figure 4.3. The *core*

[1]In our system, users specify the symmetric, periodic K-availability of agents (of the form $\mathscr{A}^S(A,B,K)$), as described in Section 3.2. The schedule is generated based on such availabilities.

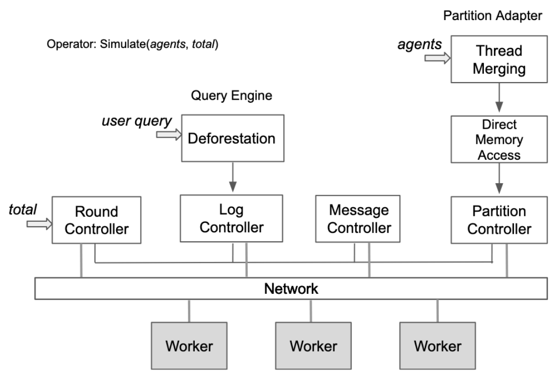


Figure 4.2: System architecture of a scalable, distributed simulation engine

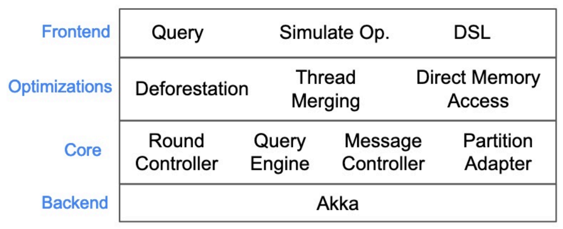


Figure 4.3: Software stack of the system. The *core* refers to the distributed engine. Users interact with the core indirectly via the programming model in the *frontend*. The applicable software optimizations for the distributed engine are summarized in the layer *optimizations*. The *backend* refers to the low-level distributed library used in our implementation.

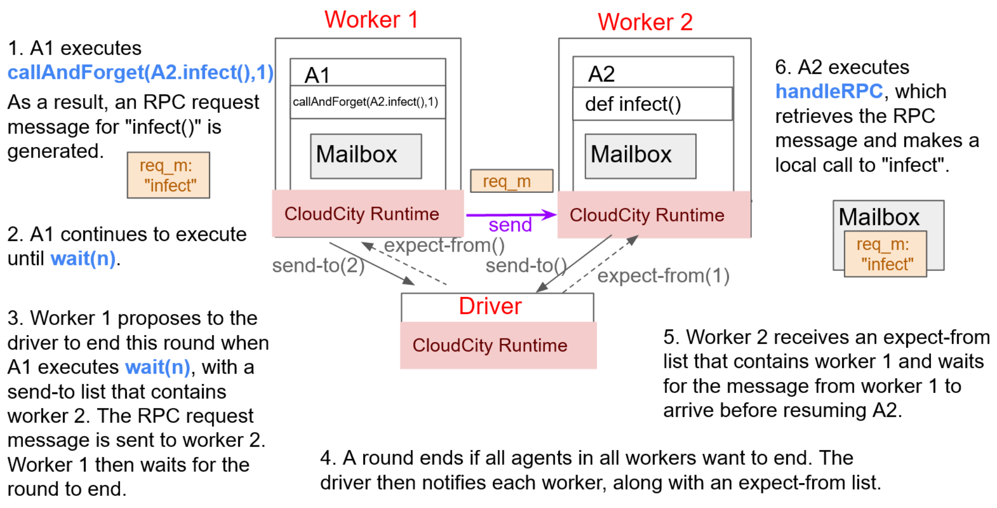
refers to the distributed engine. Users interact with the core via the programming model in the *frontend*. The DSL instructions in the programming model are compiled to Scala during the code generation, which we explain in Chapter 5. The *optimizations* layer summarizes applicable system optimizations for the distributed engine (see Chapter 6). The *backend* refers to the low-level library used in our implementation. Below, we describe the four modules in the driver: round controller, query engine, message controller, and partition adapter.

4.2 Round Controller

At the beginning of every $K(S)$ rounds, the round controller in the driver 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.

4.3 Query Engine

The query engine evaluates user queries expressed in a collection-based language against the time series generated by `Simulate`, as listed in Table 2.1. If there is no user query, then this unit outputs the time series when the simulation ends. After executing $K(S)$ rounds, each



Worker 1
Worker 2
1. A1 executes callAndForget(A2.infect(),1)
As a result, an RPC request message for "infect()" is generated.
req_m: "infect"
A1
callAndForget(A2.infect(),1)
Mailbox
req_m
A2
def infect()
Mailbox
6. A2 executes handleRPC, which retrieves the RPC message and makes a local call to "infect".
CloudCity Runtime
send
CloudCity Runtime
Mailbox
req_m: "infect"
2. A1 continues to execute until wait(n).
expect-from()
send-to(2)
send-to()
expect-from(1)
Driver
CloudCity Runtime
3. Worker 1 proposes to the driver to end this round when A1 executes wait(n), with a send-to list that contains worker 2. The RPC request message is sent to worker 2. Worker 1 then waits for the round to end.
5. Worker 2 receives an expect-from list that contains worker 1 and waits for the message from worker 1 to arrive before resuming A2.
4. A round ends if all agents in all workers want to end. The driver then notifies each worker, along with an expect-from list.

Figure 4.4: A more detailed illustration, including the control flow between driver and workers, of the interaction between two agents in Figure 2.5.

worker forwards the log of its local agents over the past $K(S)$ rounds to the log controller of the driver, which aggregates logs from all workers. When applicable, the query engine performs deforestation (see Chapter 6) before materializing the log data for query evaluation.

4.4 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 Chapter 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.

4.5 Partition Adapter

The partition adapter separates agents into components, as described in Chapter 3. Per component, the partition adapter may de-parallelize agent threads to reduce resource contentions using thread merging and apply locality-based optimizations, such as direct memory accesses (which will be described later in Chapter 6).

We consider two partition strategies for the partition adapter, *static* and *dynamic*. The strategy determines whether and how agents can be dynamically re-partitioned based on runtime message analysis. Depending on the variant of partition-tolerant BSP, the partition adapter adopts different strategies.

- In $\mathscr{M}_{\mathsf{sync}}$ and $\mathscr{M}_{\mathsf{part-sync}}$, some components need to synchronize periodically at a prescribed interval, to ensure that messages with a fixed delay arrive as expected. For these two models, the partition adapter allows only the static partition strategy, separating agents exactly as specified in the partition configuration.
- In $\mathscr{M}_{\mathsf{async}}$, there is no such restriction on synchronization frequency a priori for any agents, and messages can be arbitrarily delayed and reordered. Hence, the partition adapter allows both static and dynamic partition strategies.

Supporting static partitioning in the partition adapter is straightforward. However, facilitating dynamic partition requires the system to collect message statistics, construct a communication graph for each worker, and analyze communication graphs to determine how to re-partition agents. We describe the dynamic partitioning strategy in detail in Appendix B.

5 Compiler

CloudCity provides a compiler that translates agent definitions written in our DSL into Scala. We integrate different phases of compilation, such as syntax analysis, semantic analysis, optimizations, and code generation into a single, streamlined process by using the multi-stage programming (MSP) paradigm to design the compiler.

More specifically, we use Squid library (Parreaux et al., 2017) that provides support for class lifting (Parreaux & Shaikhha, 2020), which transforms agent class definitions using our DSL into a lifted representation that can be further optimized before being compiled into executable code. We summarize the flow of our compiler in Figure 5.1. The input is a user program written in our DSL (left), which is first lifted using Squid into A-normal form (ANF) (Flanagan et al., 1993). Afterward, we transform the ANF intermediate representation (IR) into an executable program in plain Scala (right).

Recall from Figure 2.2 that our DSL contains three components: message-passing, RPCs, and synchronization. The message-passing instructions `send` and `receive` can be directly implemented as Scala functions; other DSL instructions are code generators that produce specialized instructions based on static analysis of agent class definitions.

For instance, consider the instruction `callAndForget`$(n.contact(), 1)$, where n is an instance of the agent type `Person` and $contact$ is a public method defined in the class definition of

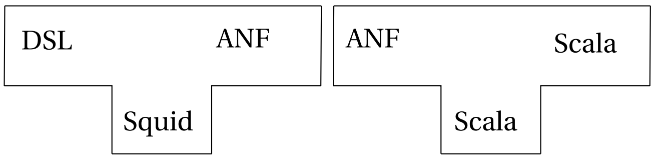


Figure 5.1: T-diagrams of CloudCity. The input is a user program written in our DSL (left), which is first lifted using Squid into the ANF representation, which is then transformed into an executable program in plain Scala (right).

```
// code pattern match in Squid
// cde is the lifted DSL
cde match {
  case code"callAndForget[$mt]({${m@MethodApplication(msg)}}:mt, $t:
  Int)" =>
    val receiverActorVar = msg.args.head.head
    val argss = msg.args.tail.map(a => a.map(arg => code"$arg"))
    val methodId = methodIdMap(msg.symbol.asTerm.name)

    // CloudCity IR that represents send
    // Generating Scala instructions is straightforward
    Send[T](receiverActorVar, methodId, t, argss)
}
```

Figure 5.2: Pseudocode of generating instructions for lifted `callAndForget` using Squid pattern match.

`Person`. In our programming model, this instruction should generate an RPC request that contains the method identifier for `contact`. To do so, the compiler needs to first derive from this instruction that the target RPC method symbol of interest is `contact` defined in the `Person` class, before looking up the corresponding identifier for this RPC method. This can be done using code pattern match in Squid, as shown in Figure 5.2. Other RPC instructions can be similarly transformed.

In this chapter, we focus on explaining how the synchronization instruction `wait` is implemented. From a high level, we can view `wait` as a coroutine instruction. Recall that an agent executing `wait(n)` waits idly until `n` rounds have passed before continuing its computation. Such behavior can be implemented using *coroutine*, which changes the control flow of a program: The execution of a program yields to the runtime system and is later resumed from the point where the program is paused. We start with a brief introduction of what coroutines are and explain how we implement `wait(n)` as coroutines, by generating an array of continuations together with the corresponding control logic for executing a continuation upon resuming.

Coroutines

The notion of coroutines was introduced in the early 1960s as a general control abstraction (Moura & Ierusalimschy, 2009). Marlin's thesis (Marlin, 1980) is widely acknowledged as a reference for coroutines, which summarized the fundamental characteristics of a coroutine as

- the local variables of a coroutine persist between successive calls, and
- the execution of a coroutine is suspended when the control leaves it. At some later time, the control re-enters the coroutine to carry on where it left off.

These characteristics stand in contrast with those of *subroutines*, such as a method defined

```
1  def execute = coroutine {() => {
2      var counter: Int = 0
3      while (true) {
4        counter += 1
5        yield(counter)
6      }
7    }
8  }
9
10 // Create a coroutine instance
11 val generator = call (execute())
12 // Execute a coroutine instance
13 generator.resume()
14 generator.value() // 1
```

Figure 5.3: Pseudocode of implementing a counter using coroutines

in an object: After a subroutine completes, all local variables declared in the subroutine are gone and can no longer be referenced; the execution of a subroutine can not be suspended or re-entered.

We illustrate the basic usage of coroutines with a simple example in Figure 5.3, implementing a counter that returns the total number of counts. On line 1, we define `execute` as a *coroutine definition* by wrapping a method body p with `coroutine(() => p)`. We can create a *coroutine instance* from the coroutine definition with `call` (line 11). To *execute* a coroutine instance, we use `resume` (line 13), which starts from the last continuation point and runs until the next `yield` (line 5), or if there is no more `yield`, then until the coroutine completes. The values of local variables in a coroutine (`counter` on line 2) are persisted between successive executions of a coroutine instance.

Implement `wait(n)` Using Coroutines

An agent that executes `wait(n)` in round t pauses the computation and waits until round $t + \mathtt{n}$ starts. Conceptually, when executing `wait(n)`, an agent proposes the number of rounds that it is waiting for (referred to as `proposeInterval`, initially `n`) 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 the round $t + \mathtt{n}$.

The control transfer between a local leader and an agent can be modeled using a `run` method and a Boolean variable `yieldFlag` (initially false) in the generated agent program after compilation, as shown in Figure 5.4. The local leader *resumes* an agent by calling the `run` method of the agent; an agent *yields* the control to the local leader when `run` terminates.

More specifically, a generated agent program contains an array of continuations (`instructions`, line 1) and an "instruction pointer" that is an index of the array (`nextInst`, line 2), initially 0.

```
 1  val instructions = Array[() => Unit]()
 2  var nextInst = 0
 3  // wait(n) changes proposeInterval and yieldFlag
 4  var proposeInterval = 1
 5  var yieldFlag = false
 6
 7  // wait(1)
 8  instructions(0) = (() => {
 9    proposeInterval = 1
10    yieldFlag = true
11    nextInst = 1
12  })
13  ...
14
15  def run(): Int = {
16    yieldFlag = false
17    while(!yieldFlag){
18      instructions(nextInst)()
19    }
20    proposeInterval
21  }
```

Figure 5.4: Pseudocode of a generated agent program in CloudCity after compilation.

Each continuation contains a sequence of non-control instructions and updates the value of `nextInst` to specify the index of the next continuation that should be evaluated. The control logic for executing a continuation is the `run` method (line 15). To clarify, we show a sample continuation generated by `wait(1)` in lines 8 – 12.[1] `wait(n)` changes `proposeInterval` (line 4) in the same way as we have described and sets `yieldFlag` (line 5) to `true`, which causes `run` to stop.

There are many benefits to using coroutines to achieve fine-grained concurrency control than multi-threading, as we explain below.

Improved usability. The generated `run` method is similar to the `Compute` method of Pregel: both serve as an agent or vertex API that is invoked at each round by the system. However, we emphasize that `run` is generated, thus taking the burden of separating computations into different rounds off users. To illustrate, we consider a simple simulation that models the behavior of a traffic light that repeatedly iterates over three colors, showing the code skeleton (the concrete implementations for the actual behavior such as how to inform vehicles to stop are not shown, which are not important for our demonstration) in both Pregel (see Figure 5.5) and CloudCity (see Figure 5.6).

For this example, users need to separate computations for different supersteps in the vertex program in Pregel by comparing the current value of a superstep at run-time with a pre-

[1] `wait(1)` actually generates multiple continuations, such as checking whether a number of rounds have passed, which are omitted in Figure 5.4 for brevity.

```
1  class TrafficLight extends Vertex<double, void, int> {
2    public void Compute(MessageIterator* msgs) {
3      if (getSuperstep() % 3 == 0) {
4        // red
5      } else if (getSuperstep() % 3 == 1) {
6        // yellow
7      } else if (getSuperstep() % 3 == 2) {
8        // green
9      }
10   }
11 }
```

Figure 5.5: Psuedocode of describing a traffic light that transitions sequentially through the colors red, yellow, and green in Pregel.

```
1  class TrafficLight extends Actor {
2    def main(): Unit = {
3      while (true) {
4        // red
5        wait(1)
6        // yellow
7        wait(1)
8        // green
9        wait(1)
10     }
11   }
12 }
```

Figure 5.6: Psuedocode of describing a traffic light that transitions sequentially through the colors red, yellow, and green in CloudCity.

defined superstep number (lines 5–10 in Figure 5.5). In contrast, CloudCity automatically determines how computations should be separated into different rounds based on `wait(n)` (lines 5–9 in Figure 5.6). Additionally, our `wait(n)` instruction makes it easy for users to modify the behavior of a traffic light to wait for an arbitrary number of rounds between different colors instead of 1.

Improved performance. Besides better usability, the coroutine implementation of `wait(n)` has improved performance due to better scheduling. If a thread does not have a mechanism to transfer the control back to the system, then in the view of a fair scheduler in a system, each thread should have the same share of the core, alternating their execution for the same amount of time, as shown in execution 1 in Figure 5.7. But this is sub-optimal because A_1 is simply busy waiting. A more efficient schedule is for A_1 to yield the control to the system and waits to be resumed after a message has arrived, as in execution 2.

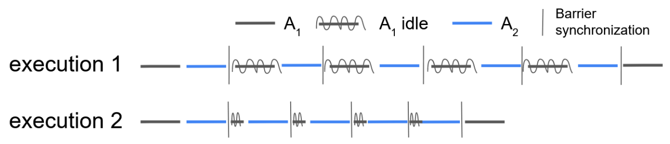


Figure 5.7: Using coroutines leads to more efficient scheduling. This example contains two long-running agent threads A_1 and A_2 competing for the same core. A_1 does some computations and waits for a message from A_2 before continuing. If an agent thread does not have a mechanism to transfer the control back to the system, then in the view of a fair scheduler in a system, each agent thread should have the same share of the core, alternating their execution for the same amount of time, as shown in execution 1. But this is sub-optimal because A_1 is simply busy waiting. A more efficient schedule is for A_1 to yield the control to the system and waits to be resumed after a message has arrived, as in execution 2.

6 Optimizations

Our system optimizations focus on decreasing the degree of concurrency through thread merging, fusing send and receive operations to reduce the overhead of message-passing, exploiting spatial information to reduce the number of agents and messages by transforming agents into tile agents through tiling, specifying a dynamic halting condition using `SimulateUntil`, and reducing intermediate data when evaluating user queries through deforestation, described below. These optimizations are designed for a single simulation. We also consider optimizations for parallel simulations, described in Appendix C.

6.1 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. The *merging order* describes how a collection of agents are merged sequentially into the same merged thread. Every merged agent is transformed into a coroutine instance. Furthermore, thread merging allows merged agents to obtain agent references for each other. The merged thread resumes the computation of merged agents sequentially as defined by the merging order in each round.

It is natural to ask why not simply use a thread pool, which is a standard approach for highly concurrent systems, such as web servers, to achieve efficient resource utilization by limiting the maximum number of active threads. We begin with a brief description of what a thread pool is and how it is used before explaining why we introduce thread merging.

In the thread pool-based approach, a system preallocates a fixed number of threads and places them in a designated *thread pool*, as demonstrated in Figure 6.1. More specifically, each thread has different states, including idle (abbreviated as label “I”) and running (abbreviated as label “R”). A thread pool is usually coupled with a *task queue*. Each thread can execute only one task

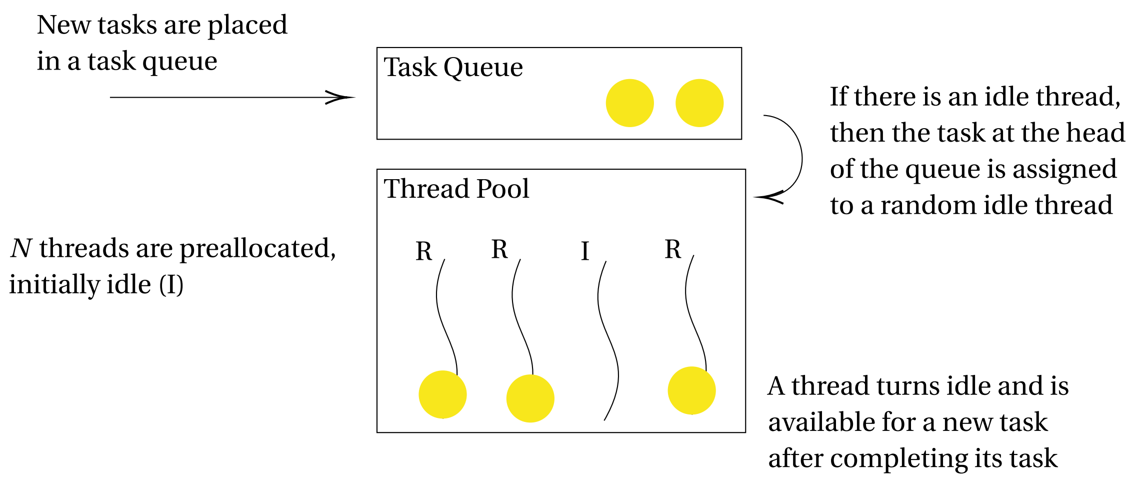

New tasks are placed in a task queue
Task Queue
If there is an idle thread, then the task at the head of the queue is assigned to a random idle thread
Thread Pool
N threads are preallocated, initially idle (I)
R
R
I
R
A thread turns idle and is available for a new task after completing its task

Figure 6.1: Illustration of how to use a thread pool to limit the maximum number of active threads in a system when executing a parallel program.

from the task queue at a time. Such threads can either be managed in a push-based approach, i.e. an idle thread is being notified when a new task has arrived and assigned the next task to execute, or a poll-based approach, where idle threads repeatedly check whether there are available tasks to be executed in the task queue and consume them if applicable. For our purpose of illustrating the differences between thread pool and thread merging, how threads in a thread pool are managed can be regarded as an unimportant low-level implementation detail. For ease of demonstration, we assume a push-based approach. When a new task arrives, it is placed in the task queue. The system checks if there is any idle thread in the thread pool. If so, the system assigns the task to an idle thread. Otherwise, the task is stored in a queue and will be executed when a thread has completed executing its task and becomes idle.

Sharing threads in a thread pool among concurrent tasks allows the system to avoid the overhead of dynamically creating and destroying threads when executing concurrent tasks. Furthermore, using a thread pool ensures that no more than a prescribed number of tasks are executed concurrently, which leads to more efficient resource utilization. Hence, thread pools are prevalent in concurrent programming, including web servers and general-purpose parallel frameworks, to manage the execution of concurrent tasks efficiently.

However, if the number of tasks in a task queue far exceeds the number of threads in a thread pool, which is configured to be the number of hardware threads on a machine for better hardware utilization, then the *task scheduling overhead* – the time between when a thread immediately becomes idle after completing a task and the time when the thread starts executing a new task again – becomes a concern. The task scheduling overhead worsens if each task takes relatively a short time to complete. In such cases, it is more desirable to combine multiple tasks into a single task for better performance, as illustrated in Figure 6.2.

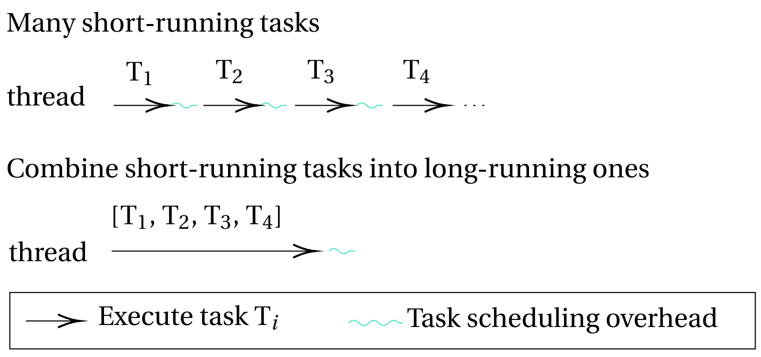


Figure 6.2: Illustration of task scheduling overhead when using a thread pool. When there are a large number of short-running tasks, it is desirable to combine such tasks into a few long-running tasks to amortize task scheduling overhead, as we do in thread merging.

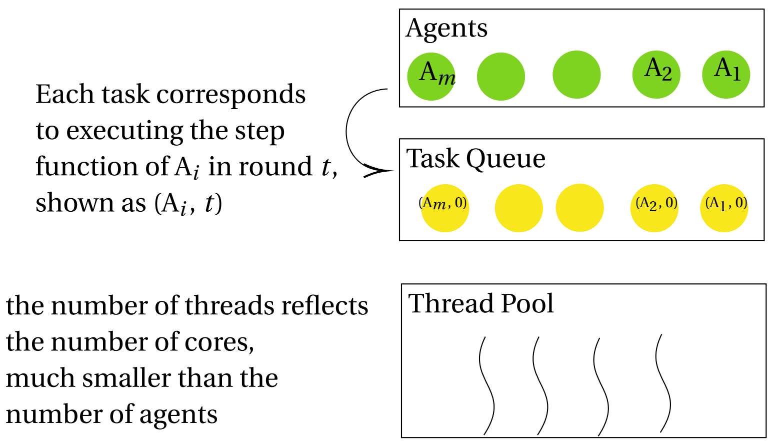
In CloudCity, each agent is by default single-threaded. Per round, an agent executes its `run` method to the completion. If we use a thread pool to share threads among agents, then each agent should obtain a thread before it executes the step function and returns the thread when the step function has been completed. In other words, a task corresponds to running the step function of an agent A_i in a round t. But the computation of an agent in a round is relatively simple and short-running, using a thread pool naively can lead to poor performance for a large number of agents, as we just explained. This can be addressed by thread merging, as shown in Figure 6.3.

While a merged thread task in Figure 6.3 can be viewed as a sequence of agent tasks, a merged thread is more than a sequence of agents. In particular, a merged thread exposes the sequential execution ordering of the merged agents to these agents. After being transformed into a coroutine object, each merged agent now contains references to other agents that share the same merged thread.[1] This allows optimizations that exploit the sequential execution of tasks, such as *direct memory accesses*, which we will explain in detail below.

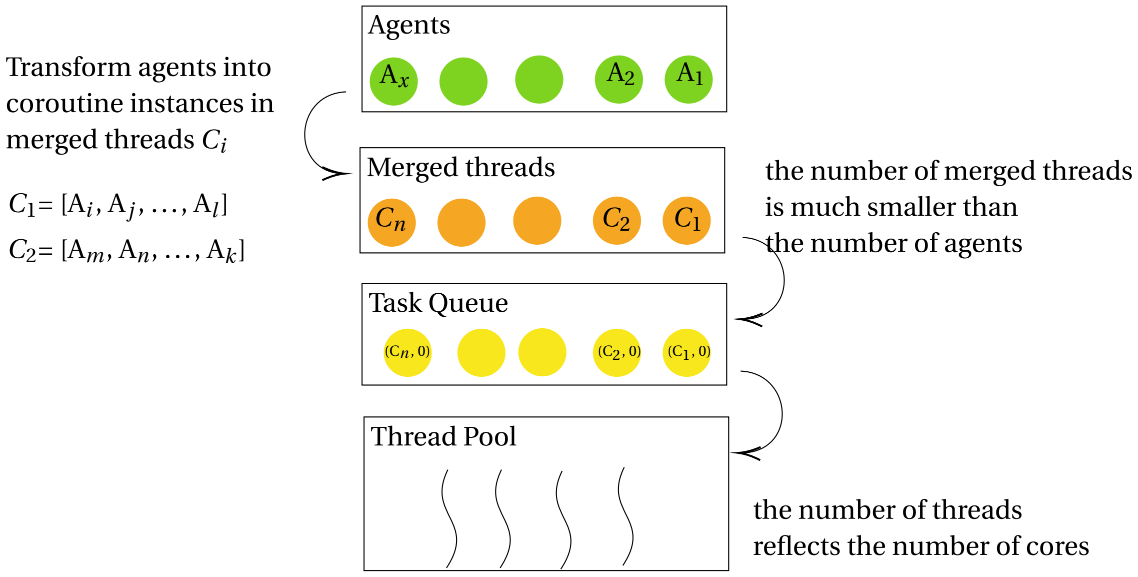
6.2 Fuse Send and Receive Operations

Message-passing is general-purpose and oblivious to how messages are processed. Sending a message always pays the computation overhead of packing a message object and the memory overhead of buffering a message. Similarly, receiving a message also has the computation overhead of unpacking a message object and deciding how to process it, such as looking up

[1]Due to shared resources and shared agent references, thread merging might raise concerns about data privacy. Thread merging is done in a way that isolates the local states of agents; states of another agent can be addressed only through messaging. The references to other agents in the same shared thread are only conveyed at runtime and cannot be accessed through the original code of the agent. Any additional privacy requirements (such as multi-owner and privacy-sensitive protocols) could be added through user libraries.


Agents
A_m
A_2
A_1
Each task corresponds to executing the step function of A_i in round t, shown as (A_i, t)
Task Queue
$(A_m, 0)$
$(A_2, 0)$
$(A_1, 0)$
the number of threads reflects the number of cores, much smaller than the number of agents
Thread Pool

(a) Before thread merging.


Agents
A_x
A_2
A_1
Transform agents into coroutine instances in merged threads C_i
$C_1 = [A_i, A_j, \ldots, A_l]$
$C_2 = [A_m, A_n, \ldots, A_k]$
Merged threads
C_n
C_2
C_1
the number of merged threads is much smaller than the number of agents
Task Queue
$(C_n, 0)$
$(C_2, 0)$
$(C_1, 0)$
Thread Pool
the number of threads reflects the number of cores

(b) After thread merging.

Figure 6.3: Illustration of how thread merging transforms a large collection of short-running agent tasks into a few long-running merged thread tasks: (a) before thread merging, and (b) after thread merging.

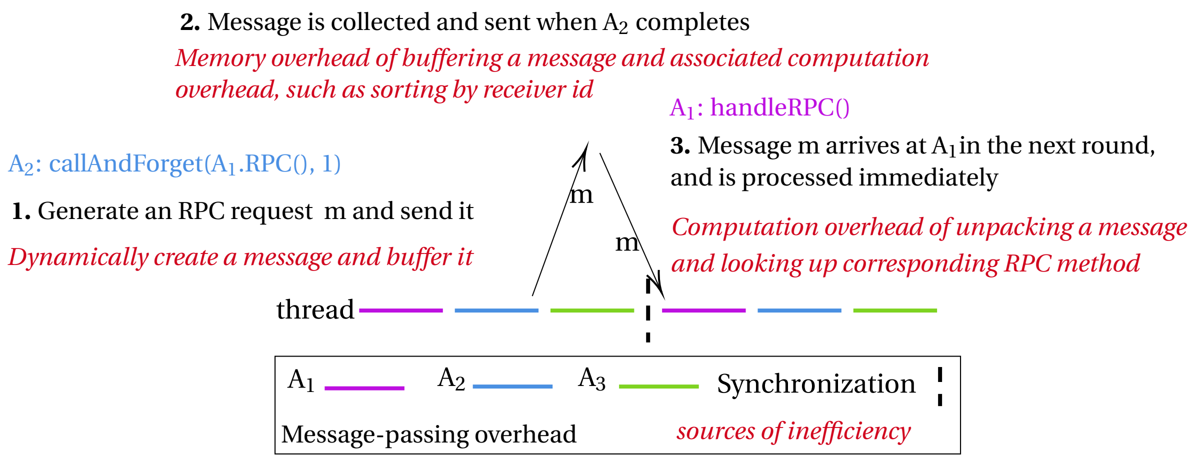

2. Message is collected and sent when A_2 completes
Memory overhead of buffering a message and associated computation overhead, such as sorting by receiver id
A_1: handleRPC()
3. Message m arrives at A_1 in the next round, and is processed immediately
Computation overhead of unpacking a message and looking up corresponding RPC method
A_2: callAndForget(A_1.RPC(), 1)
1. Generate an RPC request m and send it
Dynamically create a message and buffer it
m
m
thread
A_1
A_2
A_3
Synchronization
Message-passing overhead
sources of inefficiency

Figure 6.4: Illustration of message-passing overhead. $\mathtt{A}_1$, $\mathtt{A}_2$, and $\mathtt{A}_3$ are three agents that are executed sequentially in this order and share the same thread. $\mathtt{A}_2$ sends an RPC request message `m` to $\mathtt{A}_1$ at the end of a round with a delay of one using `callAndForget`; $\mathtt{A}_1$ processes `m` immediately at the beginning of the next round using `handleRPC`. The message-passing overhead for each step of generating, sending, and receiving `m` is explained in black, and the corresponding source of inefficiency is explained in red.

the corresponding RPC method, and possibly the memory overhead if received messages need to be buffered for delayed processing.

To illustrate, Figure 6.4 contains three agents $\mathtt{A}_1$, $\mathtt{A}_2$, and $\mathtt{A}_3$ that are executed sequentially in this order on a core. We assume that $\mathtt{A}_2$ sends an RPC request message `m` to $\mathtt{A}_1$ at the end of a round with a delay of one using `callAndForget` (shown in blue); $\mathtt{A}_1$ processes received RPC messages immediately at the beginning of each round using `handleRPC` (shown in purple). We clarify the message-passing overhead in each step of generating, sending, and receiving `m`, and explain sources of inefficiency in red.

For communication-intensive workloads, the overhead of message-passing exacerbates and can become a performance bottleneck, as is the case for agent-based simulations. Agents spend the majority of their time sending and processing messages. Reducing the overhead of message-passing can lead to significant performance benefits.

We can fuse the operations of sending and receiving messages to bypass message-passing overhead by transforming messaging to direct memory accesses (DMA). Intuitively, if a receiver thread encapsulates instructions for how to process a message in a public method, which can also be invoked by a sender thread, then the sender may invoke such a method in the receiver directly (which we refer to as direct memory accesses,[2] abbreviated as DMA). This way, a

[2]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.

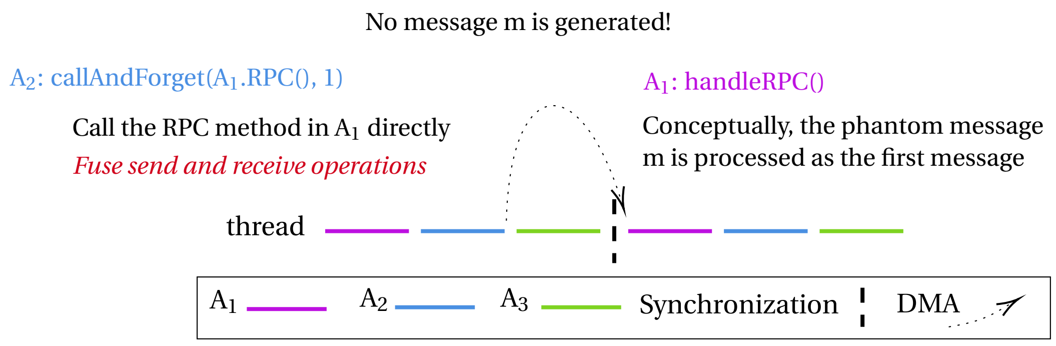

No message m is generated!
A₂: callAndForget(A₁.RPC(), 1)
Call the RPC method in A₁ directly
Fuse send and receive operations
A₁: handleRPC()
Conceptually, the phantom message m is processed as the first message
thread
A₁
A₂
A₃
Synchronization
DMA

Figure 6.5: Illustration of fusing the send and receive operations for agents in Figure 6.4. `A`$_2$ invokes `A`$_1$`.RPC()` directly, without ever generating message `m`. This allows agents to bypass message-passing overhead.

"message" can be viewed as sent immediately and processed in a single operation, effectively fusing send and receive operations.

Figure 6.5 demonstrates how to fuse send and receive operations for the previous example in Figure 6.4. If `A`$_2$ invokes `A`$_1$`.RPC()` directly, then there is no need for message `m`, avoiding the message-passing overhead entirely.

However, applying DMA naively can cause data corruption and change event orderings compared with message-passing. For our purpose, an *event* refers to an atomic execution[3] of a sequence of instructions in an agent program. The granularity of an event – in terms of the number of instructions – varies, ranging from a single instruction to a block of instructions, depending on what we want to model.

In what follows, we start by explaining naive DMA and its limitations, clarifying our assumptions and explaining basic notions that are used in the rest of this section, such as state transitions. After that, we explore different programming paradigms that make programming agents using DMA primitives more straightforward.

6.2.1 Naive DMA

In our setup, direct memory access (DMA) refers to the mechanism in which a sender agent directly invokes an RPC method defined in a receiver agent directly and waits for the completion of the RPC method's execution. This is in contrast to the message-passing approach, which requires creating a message to invoke an RPC method and then sending this message to the receiver for execution.

While the idea of DMA is straightforward and resembles calling a public method defined in a

[3]In the sense that it runs to completion without being interrupted by other agents.

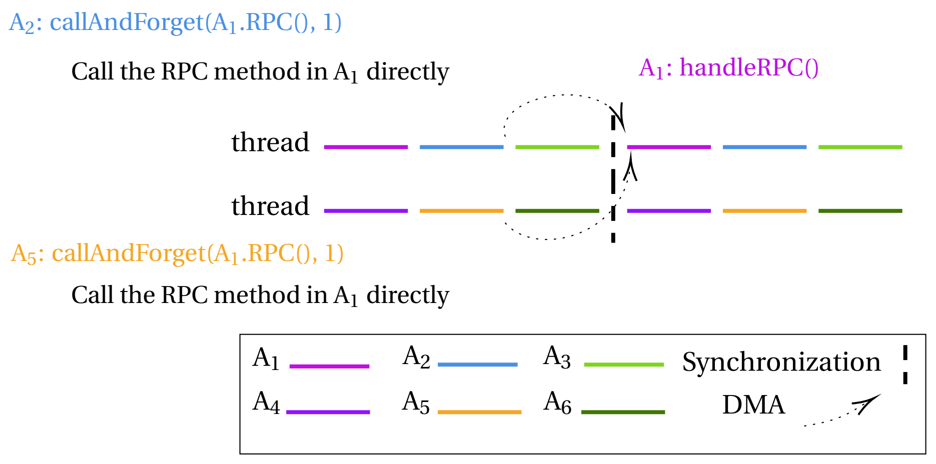

A2: callAndForget(A1.RPC(), 1)
Call the RPC method in A1 directly
A1: handleRPC()
thread
thread
A5: callAndForget(A1.RPC(), 1)
Call the RPC method in A1 directly
A1
A2
A3
Synchronization
A4
A5
A6
DMA

Figure 6.6: Illustration of how DMA can cause data corruption if invoked concurrently. Both A_2 and A_5 invoke A_1.RPC() directly, which can cause data corruption: The invocation from A_2 can accidentally overwrite a variable in A_1 that is modified concurrently by the invocation from A_5, leading to undesired data corruption.

different object in a plain old object-oriented program, there are some technical challenges when applying DMA in highly concurrent distributed applications, especially as an optimization to replace message-passing. In what follows, we examine such challenges and describe how we address them.

Atomicity

For starters, every agent is by default single-threaded. In the most general case, every agent can be executed by a different thread. Executing a method in a remote thread can cause data corruption if this method is not executed atomically: An invocation may overwrite a variable that is modified concurrently by another invocation, as shown in Figure 6.6. If both A_2 and A_5 invoke A_1.RPC(), then the invocation from A_2 can accidentally overwrite a variable in A_1 that is modified concurrently by the invocation from A_5, leading to undesired data corruption.

To address this, we restrict DMA to be applicable only if the sender shares the same thread with the receiver, as is the case for the example in Figure 6.4. Hence, for any RPC method, no concurrent DMA invocations are possible. A DMA always runs to completion atomically. In CloudCity, this requirement can be easily achieved by applying DMA only if sender and receiver agents are merged in the same thread.

Programming Model

We would like DMA to be applied as a system optimization. Syntactically, users define agent programs under the same message-passing abstraction: Agents communicate by sending messages using our DSL instructions, but during execution, the system may compile messages away and use DMA. In other words, instead of executing instructions that create and send messages, the system executes instructions that invoke remote methods directly.

Annotate RPC method with "allowDirectAccess". We let users determine whether event reordering due to fusing send and receive operations for a particular RPC method is acceptable depending on the application requirement. In particular, such event reordering does not assume any particular agent merge order, hence covering all possible event reorder where agents can be merged into any number of threads in any order. If such reordering is allowed, then users can annotate an RPC method with a modifier "allowDirectAccess".

Fuse send instruction `callAndForget`. Our compiler inserts DMA instructions statically when translating communication instructions in DSL. Recall that each public method in an agent program encapsulates the instructions for how an RPC request message should be processed. In addition, we consider fusing send and receive operations only if a sender agent does not expect to obtain a reply from the receiver. This narrows our choice down to only `callAndForget`, which does not return anything.

Figure 6.7 shows how our compiler translates `callAndForget`, where lines 8–12 are added (compared with Figure 5.2) for generating DMA instructions: line 8 ensures that only methods annotated with the modifier "allowDirectAccess" are eligible to be accessed via DMA; line 9 ensures that DMA can only be applied if both the sender and receiver agents are merged into the same thread; and line 10 generates the actual instruction that corresponds to invoking a method defined in a different object directly in Scala.

In short, for communication instruction `callAndForget` in the DSL, our system executes instructions for DMA rather than instructions for message-passing if and only if both of the following conditions are satisfied:

- the sender shares the same thread with the receiver; and
- the RPC method `receiver.API` specified in `callAndForget` is annotated with `allowDirectAccess`.

Customize merge order. Users have fine-tuned control over the merge order of agents and can decide how agents should be merged into a prescribed number of merged threads.

```
cde match {
  case code"callAndForget[$mt]({${m@MethodApplication(msg)}}:mt, $t:
  Int)" =>
    val receiverActorVar = msg.args.head.head
    val argss = msg.args.tail.map(a => a.map(arg => code"$arg"))
    val methodId = methodIdMap(msg.symbol.asTerm.name)

    // IR
    if (methodsMap(msg.symbol).compileTimeModifiers.contains("
  allowDirectAccess")){
     IfThenElse(code"$actorSelfVariable.reachableAgents.contains($
  receiverActorVar.id)",
       ScalaCode(m), // If DMA, then call directly
       Send[T](receiverActorVar, methodId, t, argss)
    } else {
      Send[T](receiverActorVar, methodId, t, argss)
    }
}
```

Figure 6.7: Pseudocode of updated `callAndForget` that generates instructions to support DMA.

Following the above discussion, it becomes apparent that for any time series of a simulation obtained where agent programs do not contain RPC methods with annotation "allowDirectAccess" (also referred to as non-annotated), this time series can also be obtained after transforming every RPC method in agent programs to include this annotation (also referred to as annotated), by simply separating each agent into a distinct thread: Even with the annotation `allowDirectAccess`, DMA instructions are not executed due to the requirement that the sender and receiver must share a thread. For the other direction, there exists time series obtained with annotated agent programs that cannot be obtained otherwise due to event reordering after fusing send and receive operations.

However, if agents are merged into pre-determined merged threads and DMA instructions are indeed executed, then it is in general not true that every time series obtained under non-annotated agent programs can also be obtained under annotated agent programs.

Observability

Fusing the send and receive operations can reorder events and possibly change what users observe compared to message-passing. We make this concrete in this section, start by clarifying some basic notions.

State Transition. We use *state transitions* to model how events change the value of agent attributes locally in an agent during a simulation: Each *state* is a data structure that contains the value of some agent attributes, and each *transition* corresponds to an event that changes a starting state to a different state, labeled with the event name.

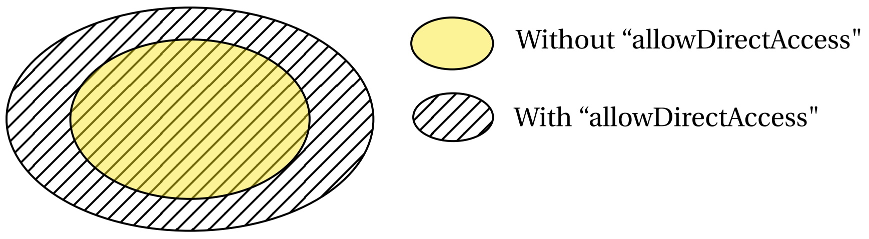


Figure 6.8: For any time series of a simulation obtained where agent programs do not contain RPC methods with annotation “allowDirectAccess”, this time series can also be obtained after transforming every RPC method in the simulation to include this annotation, by simply separating each agent into a distinct thread: Even with the annotation `allowDirectAccess`, DMA instructions are not executed due to the requirement that the sender and receiver must share a thread. For the other direction, there exists time series obtained with annotated agent programs that cannot be obtained otherwise, due to possible event reordering after fusing send and receive operations.

By default, we use the transition label D to represent a DMA event and C for calling `handleRPC`, unless specified otherwise. We point out that for simplicity, we do not distinguish DMAs from different sender agents that occur in different rounds in the label notation, though these are distinct events. For each following example where state transitions are presented, we will clarify which sender agents make DMA and when DMA occurs in the context of given cases.

Observable Agent Attributes. We assume that for any agent in a snapshot of a time series, users can only observe user-defined agent attributes, that is, agent attributes specified in the source code of an agent program before compilation. In particular, agent attributes that are introduced by our system either at compile time or run time, such as current round number, cannot be observed or queried by users. We also refer to non-observable agent attributes as *auxiliary attributes.*

On the one hand, making the distinction between observable and non-observable agent attributes allows our system to apply more aggressive optimizations without being observed by users. For instance, in generalized double-buffering, our system makes a copy of each observable agent attribute and separates the effect of DMA and message-passing by transforming the generated code for RPC methods to modify non-observable agent attributes when executing DMA and observable agent attributes when executing message-passing.

On the other hand, separating agent attributes into observable and non-observable aligns well with the intuition of how users observe a time series. For agent attributes that are added by our system, users do not know about the very existence of such variables, let alone their names, which are necessary for querying and observing them.

Indistinguishability. For a simulation, users observe the *time series* generated by an execution of the simulation, which is a sequence of snapshots. Each snapshot contains a set of all agent objects taken at the end of a round, immediately after each agent has completed its execution. This is notably different from waiting until all agents have completed their execution for the current round before taking a snapshot.

Definition 6.1. *Let S be a time series generated by a simulation with DMA enabled. If S can also be obtained with DMA disabled, such as by removing all* `allowDirectAccess` *annotations or separating each agent into a separate thread, then a user cannot distinguish whether DMA has been applied to the simulation. Otherwise, a user can distinguish.*

For a simulation, it is often desirable that users cannot distinguish whether DMA has been applied. But this is difficult to achieve, as we illustrate via the following examples.

Examples

We demonstrate the challenges of achieving indistinguishability in naive DMA through examples. We use color red to highlight values in state transitions that are observed by users. Additionally, since `wait(n)` is a control instruction that does not change the value of any user-observed states,[4] we omit `wait(n)` when illustrating how the value of agent attributes changes using state transitions.

Example 6.2. *Consider three agents,* $\mathtt{A}_1$, $\mathtt{A}_2$, *and* $\mathtt{A}_3$, *that are merged together in this order.* $\mathtt{A}_2$ *models a shared counter with a variable* `counter` *(initially 0) and an operation* `inc`*. In every round,* $\mathtt{A}_1$ *and* $\mathtt{A}_3$ *send an RPC request using* $\mathtt{callAndForget(A_1.inc(),1)}$*;* $\mathtt{A}_2$ *processes all messages. The definition of* A_2 *is shown in Figure 6.9.*

We now show that users *can* distinguish whether a time series is obtained under message-passing or DMA by observing the value of `counter` in the time series generated by Example 6.2.

Proof. We demonstrate that users observe different values for `counter` under message-passing and DMA for Example 6.2. The value of `counter` is initially 0. With message-passing, in the first snapshot, `counter` is 0, since no messages have arrived yet. For DMA, because $\mathtt{A}_1$ is merged before $\mathtt{A}_2$, $\mathtt{A}_1$ calls `inc` and modifies the value of `counter` before $\mathtt{A}_2$ runs. In the first round, the value of `counter` saved in the snapshot is 1, which is different from 0 under message-passing. Thus, users can distinguish whether a time series is generated under message-passing or DMA. □

Figure 6.10 and Figure 6.11 demonstrate how `counter` changes in the first four rounds under message-passing and DMA respectively.

[4]`wait(n)` changes the value of the local clock in an agent, but the clock is a system variable that cannot be queried or observed by users.

```
class Counter extends Agent {
  var counter: Int = 0

  @allowDirectAccess
  def inc(): Unit = {
    counter += 1
  }

  def main(): Unit = {
    while (true) {
      handleRPC()
      wait(1)
    }
  }
}
```

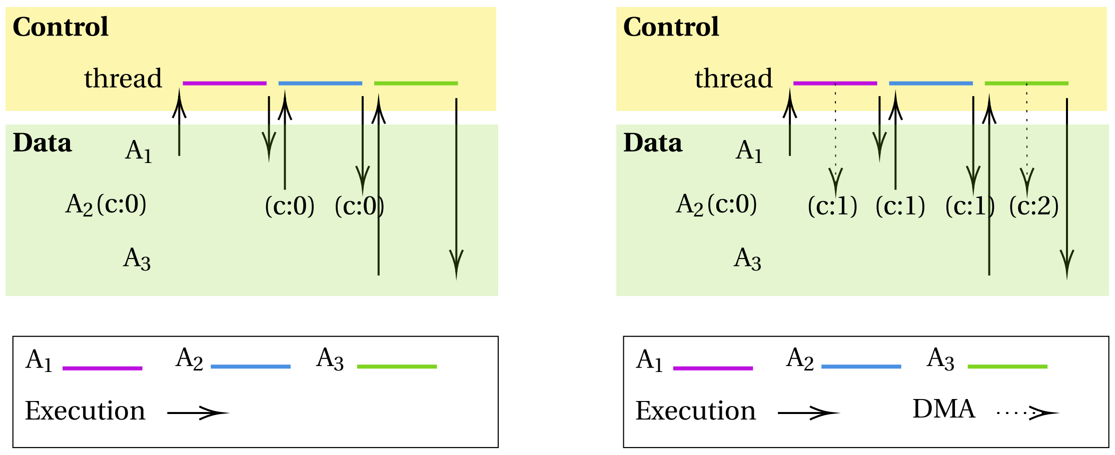
Figure 6.9: Pseudocode of `A`$_2$ definition for Example 6.2.

$$0 \xrightarrow{C} 0 \xrightarrow{C} 2 \xrightarrow{C} 4 \xrightarrow{C} 6 \cdots$$

Figure 6.10: State transition that shows how the value of `counter` changes in the first four rounds under message-passing for Example 6.2.

$$0 \xrightarrow{D} 1 \xrightarrow{C} 1 \xrightarrow{D} 2 \xrightarrow{D} 3 \xrightarrow{C} 3 \xrightarrow{D} 4 \xrightarrow{D} 5 \xrightarrow{C} 5 \xrightarrow{D} 6 \xrightarrow{D} 7 \xrightarrow{C} 7 \cdots$$

Figure 6.11: State transition that shows how the value of `counter` changes in the first four rounds under DMA for Example 6.2, where `A`$_1$, `A`$_2$ and `A`$_3$ are merged sequentially together.



(a) Message-passing. (b) DMA.

Figure 6.12: Illustration of how DMA can alter event order and change the value of a user-observable attribute `counter` (shortened as c) for Example 6.2: (a) message-passing, and (b) DMA. In the snapshot taken for the first round, c has value 0 in (a) and 1 in (b). Notably, DMA from $\mathtt{A}_3$ that changes c to 2 in (b) happens after $\mathtt{A}_2$ has completed and been saved to the snapshot, hence c still has value 1 in the snapshot.

Now consider a slight variation of Example 6.2, by merging agents differently.

Example 6.3. *This example is identical to Example 6.2 except for the merging order.* $\mathtt{A}_2$ *is now merged first, followed by* $\mathtt{A}_1$ *and* $\mathtt{A}_3$*. DMAs from* $\mathtt{A}_1$ *and* $\mathtt{A}_3$ *to* $\mathtt{A}_2$ *arrive only after* $\mathtt{A}_2$ *has completed its execution and saved to the corresponding snapshot. The agent definitions remain unchanged from Example 6.2.*

We show that users *cannot* distinguish whether a time series is generated under message-passing or DMA by observing the value of `counter` in the time series for Example 6.3. Figure 6.13 demonstrates how `counter` changes under DMA in the first four rounds in this example.

Proof. In Figure 6.9, $\mathtt{A}_2$ only reads or writes `counter` via `inc`. For simplicity, we refer to `counter` as v and `inc` as H. H can be viewed as a function of v: $H(v) = v + 1$. The value of v can be described as $H^k(v_0)$, where v_0 is the initial value of v, 0 in this example, and k is the number of times that H has been invoked so far since a simulation has started, by either DMA or processing an RPC request message that calls H.

Users observe the same value for v under both message-passing and DMA if and only if, for any snapshot taken in round t, the number of times that H has been invoked is the same in both cases.

$$0 \xrightarrow{C} 0 \xrightarrow{D} 1 \xrightarrow{D} 2 \xrightarrow{C} 2 \xrightarrow{D} 3 \xrightarrow{D} 4 \xrightarrow{C} 4 \xrightarrow{D} 5 \xrightarrow{D} 6 \xrightarrow{C} 6 \cdots.$$

Figure 6.13: State transition that shows how the value of `counter` changes in the first four rounds under DMA for Example 6.3, where $\texttt{A}_2, \texttt{A}_1$ and $\texttt{A}_3$ are merged sequentially together.

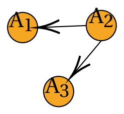


Figure 6.14: DMA graph for Example 6.2. Every vertex represents an agent. For each agent that uses `callAndForget` in its agent program, a directed edge is added from the *receiver* to the *sender*, if not already present. Topological sorts for this DAG include $\texttt{A}_2, \texttt{A}_1, \texttt{A}_3$ and $\texttt{A}_2, \texttt{A}_3, \texttt{A}_1$.

For message-passing, a message sent in round t will arrive and invoke H (upon processing) in round $t+1$, since message latency is 1 and $\texttt{A}_2$ calls `handleRPC` at the beginning of each round. Since there are two remote agents and each sends one message to $\texttt{A}_2$ per round, the number of times that H is invoked in $\texttt{A}_2$ for the snapshot in round $t (t \geq 0)$ is $2t$.

For DMA, the number of times that H is invoked depends on the number of sender agents that are merged before $\texttt{A}_2$, because in round t each of such sender agents starts executing round t before $\texttt{A}_2$ starts and invokes H immediately. In Example 6.3, $\texttt{A}_2$ is merged first, thus the number of times that H has been invoked remains the same in both message-passing and DMA. Therefore, users cannot distinguish whether a time series is generated under message-passing or DMA for Example 6.2. □

DMA Graph. Examples 6.2 and 6.3 demonstrate that what users observe in a time series can depend on how agents are merged. We explain this dependency using *DMA graph*, which is a directed graph that models how agents communicate using `callAndForget`: Every agent is added as a vertex to the DMA graph. For each agent that uses `callAndForget` in its agent program, a directed edge is added from the *receiver* to the *sender*, if not already present. To clarify, Figure 6.14 presents the DMA graph constructed for Example 6.2. While adding an edge that points from the receiver to the sender can appear counter-intuitive, it will become clear soon when we discuss the topological sorts of a DMA graph.

A *topological sort* of a directed graph is a linear ordering of all vertices in the graph, such that for every directed edge $e_{i,j}$ from vertex i to j, i is ordered before j. A topological sort exists if and only if the directed graph is acyclic (DAG). A DAG can have more than one topological sort. In particular, the DMA graph for Example 6.2 shown in Figure 6.14 is acyclic; both $\texttt{A}_2, \texttt{A}_1, \texttt{A}_3$ and $\texttt{A}_2, \texttt{A}_3, \texttt{A}_1$ are topological sorts for Figure 6.14.

Notice that the topological sort `A`$_2$, `A`$_1$, `A`$_3$ for Figure 6.14 is also the merge order of agents in Example 6.3, in which users observe the same time series under both DMA and message-passing. This is not a coincidence, as we explain below.

Let all messages in a simulation have latency 1 and each agent calls `handleRPC` at the beginning of every round. We show that for any time series, if a topological sort of the DMA graph for the agents exists and agents are merged according to a topological sort, then users cannot distinguish whether a time series is generated under message-passing or under DMA.

Proof. For message-passing, a message that is sent in round t to agent `A` for an RPC method H will be executed at the beginning of the following round $t+1$, since all messages have latency 1 and agents call `handleRPC` at the beginning of each round.

To be indistinguishable from message-passing, we need to show that the same is true for DMA: DMA that invokes H defined in `A` in round t will conceptually only be executed in round $t+1$. While this sounds contradicting to the promise of the immediate execution of DMA, this in fact follows directly from agents being executed sequentially according to a topological ordering of the DMA graph after merging. In round t, an agent that sends a DMA to `A` starts executing only after `A` has completed and saved to the snapshot. Hence, from `A`'s perspective, a DMA sent from a remote agent in round t only invokes H in the beginning of round $t+1$, in the same way as message-passing. □

For agents that are merged sequentially according to a topological ordering, a DMA can be viewed as a *partial evaluation of*`handleRPC`, which processes a corresponding message that otherwise will be created, sent, and received under message-passing.

To summarize, in this section, we have

- motivated why fusing send and receive operations is a promising optimization that can lead to considerable speedup for agent-based simulations; and
- explained technical challenges, including atomicity, programming model, and observability, when applying fusion in highly concurrent applications, and described how we address these challenges.

In particular, we have highlighted risks when applying fusion, such as event reordering that can possibly generate a time series that is not otherwise possible under message-passing, through examples Example 6.2 and Example 6.3. Based on the application requirement, users specify whether such arbitrary event reordering due to fusion is allowed, and if so, annotate the corresponding RPC methods with "allowDirectAccess", a primitive provided in our programming model.

We have also examined under what condition is the indistinguishability result satisfied. Specifically, we prove that if the DMA graph of a simulation has a topological sort and agents are merged according to a topological sort of the DMA graph, then users cannot distinguish whether a time series is generated under DMA or message passing, given that all messages have latency 1 and all agents call `handleRPC` at the beginning of each round.

Subsequently, we explore different programming paradigms that make programming agents using primitives for naive DMA, including `callAndForget` and "allowDirectAccess", more straightforward. Notably, we examine the aforementioned example Example 6.2, which does not satisfy the indistinguishability result because agents are not merged according to a topological ordering, in the context of each programming paradigm, and show how the indistinguishability result is now satisfied.

6.2.2 Multi-Versioning DMA

Multi-versioning DMA is centered around the notion of versioned variables. Rather than assuming that there is a single version of an agent attribute and DMAs modify this attribute in situ when being executed, multi-versioning DMA assumes that agent attributes are versioned, and invocations now need to provide a desired version number before updating the value of an agent variable. For ease of explanation, in any round t, we refer to the value of a versioned variable for version number t as the *visible value* of a versioned variable. This ensures that a DMA from a remote agent in round t with version number $t+1$ will only modify agent attributes with version $t+1$, irrespective of whether this DMA is executed before or after the receiver agent has completed its execution for round t.

Computation instructions that involve versioned variables are updated accordingly. An RPC method now takes an additional input argument that denotes a version number and only updates the value of agent attributes for the specified version. Sender agents pass the same version number as an input argument, regardless of whether making a DMA or sending a message. Multi-versioning can decouple the actual execution time of a DMA from the expected execution time of a DMA that is required to be indistinguishable from message-passing.

To clarify, we examine Example 6.2 in the context of multi-versioning DMA in Example 6.4.

Example 6.4. *This example is a multi-versioning variant of Example 6.2. Unlike Example 6.2,* `counter` *is a versioned variable, and* `inc` *takes a version number* `v` *as an input argument and updates only the value of the corresponding version. In round t, the outdated version of* `counter` *with key $t-1$ is removed.* $\texttt{A}_1$ *and* $\texttt{A}_3$ *now pass a version number when sending messages to* $\texttt{A}_2$ *using* $\texttt{callAndForget(A}_2\texttt{.inc(time + 1), 1)}$*, where* `time` *is the local clock time of a sender. The merging order is* $\texttt{A}_1, \texttt{A}_2, \texttt{A}_3$.

Figure 6.17 shows the definition of $\texttt{A}_2$ for Example 6.4. The agent attribute `counter` is a versioned variable (line 2), which is implemented by a hash map, where version numbers are stored as keys. The RPC method `inc` has been updated to take a version number `v` as an input argument (lines 5–7) and update only the value of the corresponding version (line 6).

$$\begin{aligned}\{0:0\} &\xrightarrow{D} \{0:0,1:1\} \xrightarrow{C} \{0:0,1:1\} \xrightarrow{R} \{0:\textcolor{red}{0},1:1\} \xrightarrow{D} \{0:0,1:2\} \\ &\xrightarrow{D} \{0:0,1:2,2:3\} \xrightarrow{C} \{0:0,1:2,2:3\} \xrightarrow{R} \{1:\textcolor{red}{2},2:3\} \xrightarrow{D} \{1:2,2:4\} \\ &\xrightarrow{D} \{1:2,2:4,3:5\} \xrightarrow{C} \{1:2,2:4,3:5\} \xrightarrow{R} \{2:\textcolor{red}{4},3:5\} \xrightarrow{D} \{2:4,3:6\}\cdots \end{aligned} \tag{6.5}$$

Figure 6.15: State transition that shows how the value of a multi-versioned variable `counter` changes in the first three rounds under DMA in Example 6.4.

$$\{0:0\} \xrightarrow{C} \{0:0\} \xrightarrow{R} \{0:\textcolor{red}{0}\} \xrightarrow{C} \{0:0,1:2\} \xrightarrow{R} \{1:\textcolor{red}{2}\} \xrightarrow{C} \{1:2,2:4\} \xrightarrow{R} \{2:\textcolor{red}{4}\}\cdots$$

Figure 6.16: State transition that shows how the value of a multi-versioned variable `counter` changes in the first three rounds under message-passing in Example 6.4.

Since there is no other local computation in `main` that writes `counter`, we garbage-collect the outdated version at the end of each round (lines 12-15).

More concretely, the state transition in Figure 6.15 and 6.16 show how the value of `counter` changes in the first three rounds in Example 6.4, with user-observed value in the snapshot highlighted in red. We use label "R" to denote the local computation that removes an old version (lines 12-15). The visible value of `counter` in a snapshot is highlighted in red after R.

Clearly, if users can observe *all* versions of a versioned variable, then Figure 6.15, which describes how `counter` changes under DMA, can be easily distinguished from that of message-passing in Figure 6.16, which only updates the visible value. *In this and the following examples, for the indistinguishability result, we assume that users only observe visible values of versioned agent attributes.*

We now prove that in Example 6.4, users are unable to distinguish whether a time series of a simulation is obtained under DMA or message-passing.

Proof. We abbreviate the visible value of multi-versioned `counter` for version number t as v_t, and `inc` as H. The value of v_t in Example 6.4 can be described as $v_t = k_t + v_{t-1}$, where $v_0 = 0$ and k_t is the number of times that H has been invoked with input argument t, by either DMA or processing an RPC request message that calls H. Then users observe the same v_t under both message-passing and DMA if and only if k_t is the same in both cases.

This is straightforward to see for multi-versioning, because DMAs that are sent in round t by $\mathtt{A}_1$ and $\mathtt{A}_3$ in Example 6.4 have input arguments $t+1$, regardless of whether senders are merged before or after $\mathtt{A}_2$. Executing such DMAs in round t only increases the number of times that H has been invoked for a future version $t+1$, in the same way as sending messages with delay 1 in round t that get processed by `handleRPC` at the beginning of round $t+1$. □

```
1  class Counter extends Agent {
2    val counter = Map[Int, Int](0 -> 0)
3
4    @allowDirectAccess
5    def inc(v: Int): Unit = {
6      counter(v) = 1 + counter.getOrElse(v, counter(v-1))
7    }
8
9    def main(): Unit = {
10     while (true) {
11       handleRPC()
12       // drop the old version of counter
13       if (time > 0) {
14         counter.remove(time - 1)
15       }
16       wait(1)
17     }
18   }
19 }
```

Figure 6.17: Pseudocode of A_2 in Example 6.4, which is implemented using multi-versioning.

Limitations. In the agent definition in Figure 6.17 for Example 6.4, a new version of `counter` is initialized with the value of the previous version (line 6). This is application-dependent and exploits the fact that once a new version of `counter` is created in a new round, previous versions of `counter`, possibly including the currently visible version, no longer change.

But the assumption that the value of an old version is read-only once a more recent (higher-numbered) version is created is in general not true. For instance, if `main` modifies `counter` in round t, then this can change the currently visible version t after a new version $t+1$ may have already been created and updated due to DMAs.

There are several reasons why the value of an old (not the most recent) version t in a versioned variable can be modified. Above all, an agent can also receive messages from sender agents that are not merged into the same thread as itself. Consequently, when processing messages from such agents in round t, the receiver agent is updating the value of `counter` for the currently visible version t. But if there are DMAs from sender agents merged before, then a more recent version of `counter` with version number $t+1$ has already been created and updated, prior to the receiver agent processing messages. In addition, it is also common for `main` to contain other local computations over currently visible agent attributes.

To clarify application-dependent limitations of the synchronization method of Example 6.4, where a new version is created based on the value of a previous version, we consider the following two examples, Example 6.6 and Example 6.7, both modifying the value of the currently visible version of `counter` after a more recent version of `counter` has been created due to DMAs, because of processing messages and performing other local computations respectively. In both of the modified examples, we show that users *can* distinguish if a time

$$
\begin{aligned}
\{0:0\} &\xrightarrow{D} \{0:0,1:1\} \xrightarrow{C} \{0:0,1:1\} \xrightarrow{R} \{0:\textcolor{red}{0},1:1\} \\
&\xrightarrow{D} \{0:0,1:1,2:2\} \xrightarrow{C} \{0:0,1:2,2:2\} \xrightarrow{R} \{1:\textcolor{red}{2},2:2\} \\
&\xrightarrow{D} \{1:2,2:2,3:3\} \xrightarrow{C} \{1:2,2:3,3:3\} \xrightarrow{R} \{2:\textcolor{red}{3},3:3\}\cdots
\end{aligned}
$$

Figure 6.18: State transition that shows how the value of `counter` changes in the first three rounds in Example 6.6 when DMA is applied. Agents $\mathtt{A}_1$ and $\mathtt{A}_2$ are merged sequentially together. $\mathtt{A}_3$ communicates with $\mathtt{A}_2$ by sending messages.

$$\{0:0\} \xrightarrow{C} \{0:0\} \xrightarrow{R} \{0:\textcolor{red}{0}\} \xrightarrow{C} \{0:0,1:2\} \xrightarrow{R} \{1:\textcolor{red}{2}\} \xrightarrow{C} \{1:2,2:4\} \xrightarrow{R} \{2:\textcolor{red}{4}\}\cdots$$

Figure 6.19: State transition that shows how the value of `counter` changes in the first three rounds in Example 6.6 under only message-passing.

series is obtained under message-passing or DMA.

For the limitations discussed in this section, we describe how they can be addressed via staged multi-versioning in the next section. There we will revisit Example 6.6 and Example 6.7, showing that users *cannot* distinguish whether a time series is generated under message-passing or DMA.

Example 6.6. *This example is identical to Example 6.4, except that only* $\mathtt{A}_1$ *and* $\mathtt{A}_2$ *are merged together sequentially.* $\mathtt{A}_3$ *sends messages to* $\mathtt{A}_2$.

For Example 6.6, we show the state transitions for DMA and message-passing in Figure 6.18 and Figure 6.19 respectively. In the third round, users observe value 3 under DMA, but 4 under message-passing, hence the indistinguishability result of Example 6.4 is no longer satisfied.

As another example, we introduce local computation inside `main`, as described in Example 6.7. The corresponding agent definition for $\mathtt{A}_2$ is presented in Figure 6.20.

Example 6.7. *This example is a slight variation of Example 6.4 by introducing additional local computation over* `counter` *in* `main`*: Every round, after processing all messages, the visible value of* `counter` *in* $\mathtt{A}_2$ *is multiplied by two.*

Similar to Example 6.6, the indistinguishability result is no longer true in Example 6.7. Figure 6.21 and Figure 6.22 show state transition that describes how `counter` changes in Example 6.7 under DMA and message-passing respectively. Users observe that the visible value of `counter` is 8 in the third round under DMA, but 12 under message-passing.

```
class Counter extends Agent {
  val counter = Map[Int, Int](0 -> 0)

  @allowDirectAccess
  def inc(v: Int): Unit = {
    counter(v) = 1 + counter.getOrElse(v, counter(v-1))
  }

  def main(): Unit = {
    while (true) {
      handleRPC()
      counter(time) = counter(time) * 2
      // drop the old version of counter
      if (time > 0) {
        counter.remove(time - 1)
      }
      wait(1)
    }
  }
}
```

Figure 6.20: Pseudocode of A$_2$ in Example 6.7, which introduces local computation in `main`.

$$
\begin{aligned}
&\{0:0\} \xrightarrow{D} \{0:0,1:1\} \xrightarrow{C} \{0:0,1:1\} \xrightarrow{L} \{0:0,1:1\} \xrightarrow{R} \{0:0,1:1\} \xrightarrow{D} \{0:0,1:2\}\\
&\quad \xrightarrow{D} \{0:0,1:2,2:3\} \xrightarrow{C} \{0:0,1:2,2:3\} \xrightarrow{L} \{0:0,1:4,2:3\} \xrightarrow{R} \{1:4,2:3\} \xrightarrow{D} \{1:4,2:4\}\\
&\quad \xrightarrow{D} \{1:4,2:4,3:5\} \xrightarrow{C} \{1:4,2:4,3:5\} \xrightarrow{L} \{1:4,2:8,3:5\} \xrightarrow{R} \{2:8,3:5\} \xrightarrow{D} \{2:8,3:6\}\cdots
\end{aligned}
$$

Figure 6.21: State transition that shows how the value of `counter` changes in the first three rounds in Example 6.7 under DMA. Agents are merged sequentially according to the merge order A$_1$, A$_2$, and A$_3$.

$$
\begin{aligned}
&\{0:0\} \xrightarrow{C} \{0:0\} \xrightarrow{L} \{0:0\} \xrightarrow{R} \{0:0\} \xrightarrow{C} \{0:0,1:2\} \xrightarrow{L} \{0:0,1:4\} \xrightarrow{R} \{1:4\}\\
&\quad \xrightarrow{C} \{1:4,2:6\} \xrightarrow{L} \{1:4,2:12\} \xrightarrow{R} \{2:12\}\cdots
\end{aligned}
$$

Figure 6.22: State transition that shows how the value of `counter` changes in the first three rounds in Example 6.7 under message-passing.

Staged Multi-Versioning. On the one hand, the assumption that for a versioned variable, a not-the-most-recent version should be read-only and not modified again after a new version has been updated, like in Example 6.4, entirely avoids the need to synchronize between different versions after a new version has been created. Conceptually, the only "synchronization" between different versions is upon initialization: A more recent version is initialized with the same value as a previous version. This greatly simplifies the programming model of versioned variables, since users do not need to worry about providing separate synchronization methods.

On the other hand, this assumption is too strong for user-observable agent attributes, where it is common for `main` to have local computations over such variables, as seen in Example 6.6 and Example 6.7.

To address this, we propose *staged multi-versioning*. Instead of transforming a user-observable agent attribute to a versioned variable, we can use a versioned variable to stage incremental changes to an agent attribute with different version numbers when calling RPCs.

More concretely, we show how to transform Example 6.6 and Example 6.7 to use staged multi-versioning, as illustrated in Example 6.8 and Example 6.9 respectively, and show that the indistinguishability result is now again true for both examples.

Example 6.8. *This example is a staged multi-versioning variant of Example 6.6.* `A`$_2$ *has two attributes, where* `counter` *is an integer and* `delta` *is a versioned variable. The RPC method* `inc` *takes a version number and increments the corresponding value of* `delta` *for a given version.*

Separately, `A`$_2$ *aggregates staged updates visible in the current round and combines it with* `counter` *after calling* `handleRPC` *in each round.*

Figure 6.23 shows the definition of `A`$_2$ in Example 6.8. Line 3 defines a new versioned variable `delta` that stages incremental changes to `counter` when calling `inc`. The `counter` variable is a plain integer (line 2). The RPC method `inc` stages updates to the corresponding version in `delta` and does not modify `counter` directly (lines 6–8). In each round, `A`$_2$ aggregates staged updates visible in the current round and combines it with `counter` (line 14) after executing `handleRPC`. Outdated versions of `delta` are removed when no longer needed (lines 18–20).

In the state transitions that describe how `counter` changes, we introduce a new event label S for aggregating the staged updates and merging with an agent attribute, as defined in line 14 in Figure 6.23. We assume that users observe only the value of `counter`, and highlight its value at the end of each round in red.

We now show that in Example 6.8, users cannot distinguish whether a time series is generated under DMA or message-passing.

Proof. We abbreviate the multi-versioned variable `delta` as d_t, where t denotes a version number, `counter` as v, and `inc` as H. Since d_t is initially 0 for any t, the value of d_t equals the

```
class Counter extends Agent {
  var counter: Int = 0
  val delta = Map[Int, Int](0 -> 0)

  @allowDirectAccess
  def inc(v: Int): Unit = {
    delta(v) = 1 + delta.getOrElse(v, 0)
  }

  def main(): Unit = {
    while (true) {
      handleRPC()
      // apply staged updates to counter
      counter += delta(time)
      // drop old versions
      if (time > 0) {
        delta.remove(time - 1)
      }
      wait(1)
    }
  }
}
```

Figure 6.23: Pseudocode of A₂ in Example 6.9, a staged multi-versioning variant of Example 6.6.

$$(\{0:0\},0) \xrightarrow{D} (\{0:0,1:1\},0) \xrightarrow{C} (\{0:0,1:1\},0) \xrightarrow{S} (\{0:0,1:1\},0) \xrightarrow{R} (\{0:0,1:1\},0)$$
$$\xrightarrow{D} (\{0:0,1:1,2:1\},0) \xrightarrow{C} (\{0:0,1:2,2:1\},0) \xrightarrow{S} (\{0:0,1:2,2:1\},2) \xrightarrow{R} (\{1:2,2:1\},2)$$
$$\xrightarrow{D} (\{1:2,2:1,3:1\},2) \xrightarrow{C} (\{1:2,2:2,3:1\},2) \xrightarrow{S} (\{1:2,2:2,3:1\},4) \xrightarrow{R} (\{2:2,3:1\},4)\cdots$$

Figure 6.24: State transition that shows how the value of `counter` changes in the first three rounds under DMA in Example 6.8. A_1 and A_2 are merged sequentially; A_3 communicates with A_2 by sending messages.

$$(\{0:0\},0) \xrightarrow{C} (\{0:0\},0) \xrightarrow{S} (\{0:0\},0) \xrightarrow{R} (\{0:0\},0)$$
$$\xrightarrow{C} (\{0:0,1:2\},0) \xrightarrow{S} (\{0:0,1:2\},2) \xrightarrow{R} (\{1:2\},2)$$
$$\xrightarrow{C} (\{1:2,2:2\},2) \xrightarrow{S} (\{1:2,2:2\},4) \xrightarrow{R} (\{2:2\},4)\cdots$$

Figure 6.25: State transition that shows how the value of `counter` changes in the first three rounds under message-passing in Example 6.8.

```
1  class Counter extends Agent {
2    var counter: Int = 0
3    val delta = Map[Int, Int](0 -> 0)
4
5    @allowDirectAccess
6    def inc(v: Int): Unit = {
7      delta(v) = 1 + delta.getOrElse(v, 0)
8    }
9
10   def main(): Unit = {
11     while (true) {
12       handleRPC()
13       // apply staged updates to counter
14       counter += delta(time)
15       // local computation
16       counter = counter * 2
17       // drop old versions
18       if (time > 0) {
19         delta.remove(time - 1)
20       }
21       wait(1)
22     }
23   }
24 }
```

Figure 6.26: Pseudocode of `A`$_2$ in Example 6.9, a staged multi-versioning variant of Example 6.7.

number of times that H has been invoked with input (version number) t. Therefore, the value of v in the snapshot for round t (v_t) can be described as $v_t = d_t + v_{t-1}$, where $v_0 = 0$.

Users observe the same value for v under message-passing and DMA if and only if, for any snapshot in round t, d_t has the same value in both cases. The proof for this is identical to our previous proof for Figure 6.17, with the only difference that the number of times that H is invoked with version number t is denoted as d_t instead of k_t, hence omitted here. □

Example 6.9. *This example is a staged multi-versioning variant of Example 6.7. Compared with Example 6.8, the only difference of this example is that* `A`$_2$ *doubles the value of* `counter` *after aggregating changes.*

Figure 6.26 shows the definition of `A`$_2$ in Example 6.9, where line 16 contains newly added local computation that doubles the value of `counter`. Figure 6.27 and Figure 6.28 model how the value of `counter` changes under DMA and message-passing respectively in Example 6.9. In the first three rounds, users observe the same values.

Below we prove that users cannot distinguish whether a time series is obtained under message-passing or DMA for Example 6.9.

Proof. The proof is identical to Example 6.8, except that the value of v in the snapshot of t is described as $v_t = 2(d_t + v_{t-1})$, notice the constant 2 that is being multiplied, which corresponds

$$\begin{aligned}
&(\{0:0\},0) \xrightarrow{D} (\{0:0,1:1\},0) \xrightarrow{C} (\{0:0,1:1\},0) \xrightarrow{S} (\{0:0,1:1\},0) \xrightarrow{L} (\{0:0,1:1\},0) \\
&\xrightarrow{R} (\{0:0,1:1\},0) \xrightarrow{D} (\{0:0,1:2\},0) \xrightarrow{D} (\{0:0,1:2,2:1\},0) \xrightarrow{C} (\{0:0,1:2,2:1\},0) \\
&\xrightarrow{S} (\{0:0,1:2,2:1\},2) \xrightarrow{L} (\{0:0,1:4,2:3\},4) \xrightarrow{R} (\{1:2,2:1\},4) \xrightarrow{D} (\{1:2,2:2\},4) \\
&\xrightarrow{D} (\{1:2,2:2,3:1\},4) \xrightarrow{C} (\{1:2,2:2,3:1\},4) \xrightarrow{S} (\{1:2,2:2,3:1\},6) \\
&\xrightarrow{L} (\{1:2,2:2,3:1\},12) \xrightarrow{R} (\{2:2,3:1\},12) \cdots
\end{aligned}$$

Figure 6.27: State transition that shows how the value of `counter` changes in the first three rounds under DMA in Example 6.9. `A`$_1$ and `A`$_2$ are merged sequentially; `A`$_3$ communicates with `A`$_2$ by sending messages.

$$\begin{aligned}
&(\{0:0\},0) \xrightarrow{C} (\{0:0\},0) \xrightarrow{S} (\{0:0\},0) \xrightarrow{L} (\{0:0\},0) \xrightarrow{R} (\{0:0\},0) \\
&\xrightarrow{C} (\{0:0,1:2\},0) \xrightarrow{S} (\{0:0,1:2\},2) \xrightarrow{L} (\{0:0,1:2\},4) \xrightarrow{R} (\{1:2\},4) \\
&\xrightarrow{C} (\{1:2,2:2\},4) \xrightarrow{S} (\{1:2,2:2\},6) \xrightarrow{L} (\{1:2,2:2\},12) \xrightarrow{R} (\{2:2\},12) \cdots
\end{aligned}$$

Figure 6.28: State transition that shows how the value of `counter` changes in the first three rounds under message-passing in Example 6.9.

to local computation added in Example 6.9. □

6.2.3 Generalized Double-Buffering DMA

We have shown how multi-versioning DMA and its variant staged multi-versioning DMA can be used to implement Example 6.2 in a way that ensures users cannot distinguish whether a time series is generated under message-passing or DMA respectively. In this section, we describe another programming paradigm, generalized double-buffering DMA.

The intuition is that since messages take exactly one round to arrive and are processed immediately at the beginning of each round, DMAs from sender agents that are merged before `A`$_2$ need to be delayed by precisely one round, hence only two versions are needed.

More precisely, recall that in Example 6.4, the visible value of `counter` in round t observed by users can be described as $v_t = k_t + v_{t-1}$, where k_t is the number of times that `inc` is invoked with input argument t. For generalized double-buffering with only two versions, v_t can now be described as $v_{h(t)} = k_t + v_{h(t-1)}$, where $h(t) = t \bmod 2$.

Example 6.10. *This example is identical to Example 6.4, except that* `counter` *is restricted to have only two versions with keys 0 and 1 respectively. The RPC method* `inc` *is updated accordingly, by checking whether the passed version is even or odd and update the corresponding version of* `counter`*. Sender agents* `A`$_1$ *and* `A`$_3$ *remain unchanged from Example 6.4.*

```
1  class Counter extends Agent {
2    // counter has only 2 versions with key 0, 1
3    val counter = Map[Int, Int](0 -> 0)
4
5    @allowDirectAccess
6    def inc(v: Int): Unit = {
7      counter(v % 2) = 1 + counter.getOrElse(v % 2, counter((v-1) % 2))
8    }
9
10   def main(): Unit = {
11     while (true) {
12       handleRPC()
13       wait(1)
14     }
15   }
16 }
```

Figure 6.29: Pseudocode of an attempted implementation of $\mathtt{A}_2$ for Example 6.10 that uses generalized double-buffering with only two versions (0 and 1). The modulo operation is denoted with % operator. In this implementation, newer versions of `counter` are not initialized correctly (line 7) due to hash collision after modulo 2.

$$\begin{aligned}\{0:0\} &\xrightarrow{D} \{0:0,1:1\} \xrightarrow{C} \{0:0,1:1\} \xrightarrow{D} \{0:0,1:2\}\\ &\xrightarrow{D} \{0:1,1:2\} \xrightarrow{C} \{0:1,1:2\} \xrightarrow{D} \{0:2,1:2\}\\ &\xrightarrow{D} \{0:2,1:3\} \xrightarrow{C} \{0:2,1:3\} \xrightarrow{D} \{0:2,1:4\}\cdots\end{aligned}$$

Figure 6.30: State transition of how `counter` changes in Example 6.10 under DMA, if $\mathtt{A}_2$ is implemented as Figure 6.29.

At first sight, it is tempting to define $\mathtt{A}_1$ as shown in Figure 6.29, which limits the number of keys in the versioned variable `counter` to 0 and 1 (line 3), and updating the appropriate version based on whether a specified version number in the input is even or odd (lines 6–8).

But there is one caveat. In Example 6.4, a new version of `counter` is initialized with the value of the previous version (lines 5–7 in Figure 6.17). This is no longer true in Example 6.10 due to the hash collision that can arise after modulo by 2. For instance, both 0 and 2 are hashed to key 0. At the beginning of round 2, we would like version 2 to be initialized with the value of version 1. Instead, line 7 will increase the value of version 0 by 1, which is wrong and will cause users to observe that DMA has been applied, as shown in the state transition in Figure 6.30.

Another attempt is to define $\mathtt{A}_2$ as Figure 6.31, trying to fix the hash collision problem of Figure 6.29 by removing an existing version explicitly at the end of each round. Now a newer version t will be initialized with the value of the previous version $t-1$ (line 7) because the previous value with version t has been dropped. But this causes yet another problem: the removed value can contain results due to DMA in the current round. To clarify, the state

```
1  class Counter extends Agent {
2    // counter has only 2 versions with keys 0, 1
3    val counter = Map[Int, Int](0 -> 0)
4
5    @allowDirectAccess
6    def inc(v: Int): Unit = {
7      counter(v % 2) = 1 + counter.getOrElse(v % 2, counter((v-1) % 2)))
8    }
9
10   def main(): Unit = {
11     while (true) {
12       hanelRPC()
13       if (time > 0) {
14         counter.remove((time -1) % 2)
15       }
16       wait(1)
17     }
18   }
19 }
```

Figure 6.31: Pseudocode of an attempted implementation of $\mathtt{A}_2$ for Example 6.10 that uses generalized double-buffering with only two versions (0 and 1). The modulo operation is denoted with % operator. This implementation fixes the hash collision issue of Figure 6.29 by explicitly removing a previous version (lines 13–15), but now DMA results stored in `counter` can also be removed.

$$
\begin{aligned}
\{0:0\} &\xrightarrow{D} \{0:0,1:1\} \xrightarrow{C} \{0:0,1:1\} \xrightarrow{R} \{0:0\} \xrightarrow{D} \{0:0,1:1\} \\
&\xrightarrow{D} \{0:1,1:1\} \xrightarrow{C} \{0:1,1:1\} \xrightarrow{R} \{1:1\} \xrightarrow{D} \{0:2,1:1\} \\
&\xrightarrow{D} \{0:2,1:2\} \xrightarrow{C} \{0:2,1:2\} \xrightarrow{R} \{0:2\} \xrightarrow{D} \{0:2,1:3\}\cdots
\end{aligned}
$$

Figure 6.32: State transition that shows how `counter` changes under DMA for Example 6.10 where $\mathtt{A}_2$ is defined as Figure 6.31.

transition in Figure 6.32 shows how `counter` changes in Figure 6.31.

Having demonstrated why naive attempts to define $\mathtt{A}_2$ for Example 6.10 do not work, we propose different approaches to address this issue, explained below.

Separate Version Number for Synchronization from Update. Firstly, we can separate the version number that is used for checking whether synchronization should take place from the version number that is used for updating the corresponding value in a versioned variable, possibly after hashing.

Figure 6.33 shows a sample definition of $\mathtt{A}_2$ for Example 6.10 under this approach. It introduces another variable `lastSeen` (line 4) to explicitly check whether a new version should be created,

```
 1  class Counter extends Agent {
 2    // counter has only 2 versions with keys 0, 1
 3    val counter = Map[Int, Int](0 -> 0)
 4    var lastSeen: Int = 0
 5
 6    @allowDirectAccess
 7    def inc(v: Int): Unit = {
 8      if (lastSeen < v){
 9        counter(v % 2) = 1 + counter((v-1) % 2)
10        lastSeen = v
11      } else {
12        counter(v % 2) = 1 + counter(v % 2)
13      }
14    }
15
16    def main(): Unit = {
17      while (true) {
18        handleRPC()
19        wait(1)
20      }
21    }
22  }
```

Figure 6.33: Pseudocode of $\texttt{A}_2$ for Example 6.10, which separates the version number for synchronization from update.

separately from updating the value of a versioned variable `counter` (lines 7–14). In particular, `inc` checks whether an input version number v is greater than `lastSeen`, if so, a new version (after modulo 2) is created and initialized with the previous version, before increasing by one. Otherwise, the corresponding version value increments by one.

We demonstrate the state transition of how (`counter`, `lastSeen`) changes in Figure 6.33 for the first three rounds in Figure 6.34. Now users again observe a sequence of even numbers, just like under message-passing.

Now we show that users cannot distinguish whether a time series is generated under DMA or message-passing for Example 6.10, where $\texttt{A}_2$ is defined as Figure 6.33.

$$\begin{aligned}
(\{0:0\},0) &\xrightarrow{D} (\{0:0,1:1\},1) \xrightarrow{C} (\{0:0,1:1\},1) \xrightarrow{D} (\{0:0,1:2\},1)\\
&\xrightarrow{D} (\{0:3,1:2\},2) \xrightarrow{C} (\{0:3,1:2\},2) \xrightarrow{D} (\{0:4,1:2\},2)\\
&\xrightarrow{D} (\{0:4,1:5\},3) \xrightarrow{C} (\{0:4,1:5\},3) \xrightarrow{D} (\{0:4,1:6\},3)\cdots
\end{aligned}$$

Figure 6.34: State transition that shows how (`counter`, `lastSeen`) changes in the first three rounds in Example 6.10 after separating version number for synchronization from update. The definition of $\texttt{A}_2$ is shown in Figure 6.33.

```
class Counter extends Agent {
  val counter: Map[Int, Int] = Map[Int, Int](0 -> 0, 1 -> 0)

  @allowDirectAccess
  def inc(v: Int): Unit = {
    counter(v % 2) +=1
  }

  def merge(): Unit = {
    // Copy the currently updated version to the other
    if (counter(0) > counter(1)){
      counter(1) = counter(0)
    } else {
      counter(0) = counter(1)
    }
  }

  def main(): Unit = {
    while (true) {
      // System calls merge at the beginning of a round,
      handleRPC()
      wait(1)
    }
  }
}
```

Figure 6.35: Pseudocode of `A`$_2$ for Example 6.10 by introducing additional merge merge.

Proof. The value of `counter` at round t in Figure 6.33 can be described by $v_{h(t)} = k_t + v_{h(t-1)}$, where $h(t) = t \bmod 2$, k_t denotes the number of times that `inc` has been invoked with an input argument t, and $v_0 = 0$. Consequently, users cannot distinguish if a time series is generated under message-passing or DMA if and only if the value of k_t is the same for both cases. The proof for this is identical to that for Example 6.4, since sender agents `A`$_1$ and `A`$_3$ remain unchanged when transforming `A`$_2$ from multi-versioning to generalized double-buffering. □

Additional Merge Method. Alternatively, we can move the synchronization between different versions of a versioned variable to a dedicated `merge` method that is invoked at the beginning of each round. In this example, `merge` takes the maximum value of both versions of `counter` as the most recent value and updates both versions (with keys 0 and 1) to this value. The `inc` method now only updates the corresponding version of `counter` after computing the hash of an input version (modulo 2) by adding one. Note that the `merge` method is called by the system at the beginning of a round t, before any agent starts executing the given round t.

We prove below that for this example, users cannot distinguish whether a time series is obtained under DMA or message-passing, for `A`$_2$ defined as in Figure 6.35. The state transition of `A`$_2$ under DMA for the first three rounds is shown in Figure 6.36, where the label M denotes calling `merge` at the beginning of each round (except the first) by the system.

$$
\begin{aligned}
\{0:0,1:0\} &\xrightarrow{D} \{0:0,1:1\} \xrightarrow{C} \{0:0,1:1\} \xrightarrow{D} \{0:0,1:2\} \\
&\xrightarrow{M} \{0:2,1:2\} \xrightarrow{D} \{0:3,1:2\} \xrightarrow{C} \{0:3,1:2\} \xrightarrow{D} \{0:4,1:2\} \\
&\xrightarrow{M} \{0:4,1:4\} \xrightarrow{D} \{0:4,1:5\} \xrightarrow{C} \{0:4,1:5\} \xrightarrow{D} \{0:4,1:6\} \cdots .
\end{aligned}
$$

Figure 6.36: State transition that shows how `counter` changes under DMA for Example 6.10 after introducing a separate merge method.

Proof. The value of `counter` observed by users in round t can be described by $v_{h(t)} = max(v_0, v_1) + k_t$ for agent definition shown in Figure 6.35, where $h(t) = t \bmod 2$, k_t denotes the number of times that `inc` has been invoked with an input argument t, $v_0 = 0$, and $v_1 = 0$. Similarly to before, users cannot distinguish if a time series is generated under message-passing or DMA if and only if the value of k_t is the same for both cases. The proof for this is identical to that of Example 6.4 because sender agents $\mathtt{A}_1$ and $\mathtt{A}_3$ remain unchanged. □

6.3 Tiling

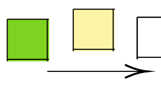
While fusing send and receive operations is one way to reduce the number of messages in a simulation, we can also exploit spatial information, when applicable, to tile an underlying graph that is embedded in a simulation, and send vector messages between tiled agents. In the scope of this thesis, we consider the technique of tiling as yet another programming paradigm, in which users define tile agents rather than cell agents when developing simulations.

For instance, we demonstrate an example that embeds a 2D grid in Figure 6.37. Assume that each cell has eight neighbors. Per round, instead of each cell sending one message to each neighbor, every tile agent sends one message to each neighbor. Depending on the size of the tile, this can reduce the number of messages in every round by orders of magnitude.

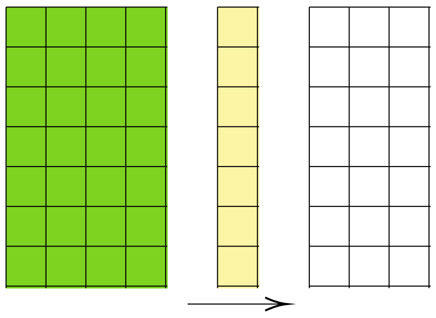
In general, tiling can be applied to any graph, as long as the structure of the graph remains static, as illustrated in Figure 6.39.

6.4 SimulateUntil Operator

Recall that the `Simulate` operator runs a simulation for a pre-determined number of `total` rounds. This semantics does not allow a simulation to terminate early, that is, fewer rounds than the specified value of `total`, based on a condition. However, in practice, this is often a desired feature. For instance, in an epidemics simulation, users may be interested in questions like “When does at least half of the population get infected?” or “When does the disease stop spreading, i.e. the number of all infected people remains unchanged for 5 consecutive rounds?”



(a) An agent represents one cell and every message contains the state of one cell.



(b) After tiling, an agent represents a tile of the 2D grid and every message contains the state of a vector of cells.

Figure 6.37: Exploit spatial information to tile agents and reduce the number of messages in a 2D grid. The representation of an agent and a message is highlighted in green and yellow respectively. (a) Under naive message-passing, a simulation shuffles and processes a large number of small messages. (b) After tiling, the number of messages that are generated and shuffled in a simulation can be reduced by orders of magnitude.

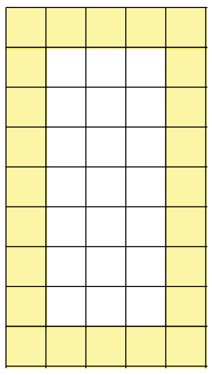


Figure 6.38: We can pad a tile agent (white) with preallocated cells (yellow) for storing vector messages received from neighbors, assuming that each cell communicates only with directly connected neighbors.

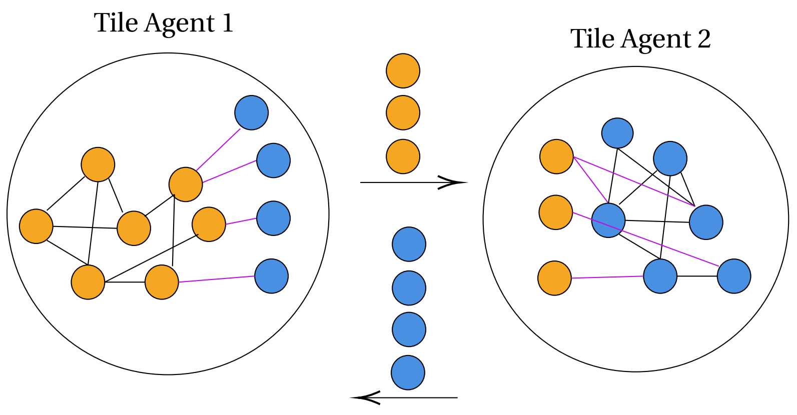


Figure 6.39: Tile agents can also be defined for a random graph, as long as the structure of the graph remains static. The vector messages exchanged between tile agents can be asymmetric. In this example, tile agent 1 sends a vector message of size 3 to tile agent 2 but expects a vector message of size 4 from tile agent 2.

For such questions, a simulation should stop and return immediately once a given condition is met. To this end, we introduce `SimulateUntil` operator:

```
SimulateUntil(agents : List[Actor], total : Int)
        (halt : (timeseries : Seq[Col[Actor]]) => Boolean).
```

Compared with `Simulate`, this operator takes an additional argument `halt : (timeseries : Seq[Col[Actor]]) => Boolean`, which is a function that returns a Boolean value upon execution. At the end of each round, `halt` function is evaluated against the current time series: If true, the simulation stops; otherwise, the simulation continues until `total` rounds have passed.

For the aforementioned questions, we can easily express them using `SimulateUntil` operator as

```
SimulateUntil(agents, 90)((ts : Seq[Col[Actor]]) => ts.last.filter(
_.asInstanceOf[Person].infectious).size) > population/2),
```

and

```
SimulateUntil(agents, 90)((ts : Seq[Col[Actor]]) => ts.takeRight(5)
.map(i => i.filter(_.asInstanceOf[Person].infectious).size).unique.size == 1),
```

respectively. It is easy to see that `Simulate` can be viewed as a special case of `SimulateUntil`, where `cond = (ts : Seq[Col[Agent]]) => false`.

6.5 Virtual Round

Sometimes there is a semantic gap between a round in our system and what users perceive as an application-level time tick. For instance, consider a minimal traffic simulation where each cell may contain at most one car. An empty cell agent can receive multiple concurrent request messages from neighboring agents, all want to move their cars to this cell. In this case, users need to implement an application-level concurrency protocol. In the simplest case, the receiver cell replies true to the first request message that is retrieved from the mailbox, and false to later messages. This can be understood as only one of the senders obtains a “lock” to this cell, and its car is allowed to move.

The application-level concurrency protocol introduces extra messages and delays. In our traffic example, a car needs to wait for an arbitrary number of rounds, induced by trying to acquire a lock, before it can actually move. Once a lock has been acquired, then the action of moving takes exactly one round. While such a concurrency protocol is necessary to ensure the semantics of a simulation, this makes it difficult, if not impossible, for users to implement high-level properties, such as the speed of a car, directly in the unit of rounds. Instead, users need to introduce a clock whose notion of time is independent of the underlying rounds and express the speed of a car in terms of this new time unit. This clock advances only after all cars have obtained their locks and are ready to move.

Depending on the application requirement, users may want some rounds, such as those induced by concurrency protocol, to not be counted. To facilitate this, we introduce *virtual rounds*, which perform a barrier synchronization to ensure that messages are sent and received, but without incrementing the value of agent clocks. Specifically, a virtual round corresponds to the instruction `wait(0)`: an agent proposes to wait for 0 round, yields the execution of control to the system, and waits to be resumed. The system takes the minimum of all the proposed rounds, 0 in this case, and adds 0 to the clock of all agents. Upon resuming, the condition on line 7 returns false, hence the agent continues its computation (line 11). We illustrate this in the pseudocode shown in Figure 6.40.

The main concern of using virtual rounds is that this can cause non-terminating simulations. In the case of livelock, one or more agents repeatedly send request messages to other agents, trying to acquire shared resources, but are always rejected and unable to make progress. Without virtual rounds, a simulation will still terminate after a prescribed total number of rounds. If agents use virtual rounds, however, a simulation may not terminate, because some agents always propose 0.

This non-terminating problem can be easily addressed by placing an upper bound on the number of consecutive virtual rounds, defaulting to the user-supplied total number of rounds. In particular, our system introduces a virtual round counter. Per round, if none of the agents has proposed 0, then this counter is cleared to 0. Otherwise, the virtual round counter increments by one. Before resuming agents, the system checks whether the virtual round counter has reached the specified upper bound, if so, the simulation terminates.

```
// wait(0)
target = time + 0
propose 0
yield

resume:
  while (time < target) {
    propose target-time
    yield
  }
  // continue
```

Figure 6.40: Pseudocode for how virtual round works, which corresponds to `wait(0)`.

Virtual rounds can also lead to the log controller of our system receiving multiple snapshots for the same round value. Only the last one of such snapshots is stored in the time series.

6.6 Deforestation

The `Simulate` operator generates a time series in the form of a sequence of snapshots of all agents (see Chapter 2). Although the time series let a user nest different simulations, such flexibility comes at the cost of a high volume of intermediate data and is not always needed. When a user query is composed with `Simulate`, we can apply deforestation (Wadler, 1988) to reduce such costs.

Deforestation eliminates intermediate data structures created when evaluating functional expressions by fusing them. For instance, consider the following functional expression

$$(1 \texttt{ to } n).\texttt{map}(i => \texttt{square}(i)).\texttt{sum},$$

evaluating this expression needs to dynamically create and destroy two collections of size n, one for (1 to n) and another generated by `map`, which is unnecessary and costly. Deforestation eliminates such intermediate data structures by transforming this functional expression into a more efficient program like

```
sum = 0
for $i$ from 1 to $n$
  sum = sum + square($i$).
```

In CloudCity, we apply deforestation to reduce the volume of time series by applying the user query to each local worker. To illustrate, we revisit Q1 in Table 2.1, which concerns how the infectious population changes as the disease progresses.

```
Simulate(agents, 90).map(snapshot
          => snapshot.filter(_.asInstanceOf[Person].infectious).size).
```

We assume the non-partitioned execution mode. Generalizing the technique presented here to partitioned executions is straightforward.

At the end of each round, a worker applies a function `m` over its local snapshot before materializing the timeseries:

```
m : Col[Agent] => Int = (snapshot : Col[Agent])
   => snapshot.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 Chapter 4). The original query has the form of

```
Simulate(agents, 90).map(m).
```

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 chapter, we assess the performance of CloudCity and compare it against current state-of-the-art distributed systems. While we also examined existing general-purpose distributed agent-based simulation engines (Coakley et al., 2012; N. Collier & North, 2011; N. T. Collier et al., 2020), such engines do not support efficient parallel computation or leave the problem of parallelization to the user (see Chapter 8). Given the synchronous round-based simulation semantics that we use in this thesis, we evaluate our system by comparing it with state-of-the-art distributed systems that support BSP-like computation: Spark (Zaharia et al., 2012), GraphX (Gonzalez et al., 2014), Giraph (Apache Giraph Developers, 2011; Ching et al., 2015), and Flink Gelly (Carbone et al., 2015; Developers, n.d.). For hardware, 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. Each server connects to routers through two 10Gb Ethernet network interface cards.

The structure of this section is as follows. In Section 7.1, we describe examples selected from representative fields where agent-based simulations are prevalent to benchmark these systems. Next, we demonstrate how the benchmark examples can be tuned to achieve different application-level outcomes in Section 7.2. Before benchmarking different systems, Section 7.3 describes important details concerning different systems when implementing the benchmark suite on such systems. In Section 7.4, we examine the suitability of different systems for running agent-based simulations using our benchmark suite. In the base case without any optimizations, our system achieves on-par performance with Giraph, whose system design is closest to CloudCity, two to ten times faster than Flink Gelly, and two to three orders of magnitude faster than Spark and GraphX, across all microbenchmark experiments, including scalability, communication frequency, and computation interval. With all optimizations enabled, our system is 10-20× faster than even Giraph, for 10,000 agents on one machine.

7.1 Benchmark Description

For a comprehensive evaluation of the efficiency of different frameworks when executing agent-based simulations, we introduce a benchmark suite that includes a diverse set of workloads for scenarios such as population dynamics, economics, and disease propagation over different social graphs, which are representative of the types of problems that agent-based simulations are commonly used to address.

7.1.1 Population Dynamics

We model the dynamics of a population modeled after Conway's Game of Life, which simulate how the population changes as people are born and existing population die due to overpopulation and underpopulation. The world is simplified to a 2D grid, where every cell is represented by an agent. Each agent is connected to eight neighboring agents on the grid. For cells on an edge of the grid, their neighbors are wrapped around on the opposing edge of the grid. Every agent has a Boolean attribute that denotes whether it is alive. Per round, each agent sends a message to every neighbor, informing them whether it is alive. Messages arrive in one round. An agent processes all received messages and updates its attribute according to the following rules:

1. If an agent is alive and the number of alive neighbors is either less than two or more than three, then the agent dies of under- or over-population respectively;
2. If an agent is dead and the number of alive neighbors is exactly three, then the agent becomes alive, which simulates reproducing new agents;
3. Otherwise, the state of an agent remains unchanged.

7.1.2 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 (Palmer et al., 1994). The stock market has one stock. The dividend per share is modeled as a stochastic variable. Both traders and stock markets are agents. The market and traders form a two-level hierarchy, where the market agent communicates with trader agents. There is no communication between traders.

In each round, every 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.

Traders are rule-based systems that aim to maximize their wealth, which consists of bank savings and stock shares. The update rule for the trader's wealth in time $t+1$ is

$$w(t+1) = (1+r)M(t) + h(t)p(t+1) + h(t)d(t+1),$$

where $M(t)$ is the amount of savings in time t, r is the interest rate, $h(t)$ is the number of stock shares, $p(t+1)$ is the new price of the stock share in time $t+1$, and $d(t+1)$ is the amount of cash dividend per share in time $t+1$.

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. For simplicity, we consider only the following five rules, which can be expanded to make the simulation more realistic, see (Palmer et al., 1994). By default, if neither buy nor sell conditions are met, then the trader agent does nothing.

- Buy if the stock dividend has increased and sell if the stock dividend has decreased.
- Buy if the 100-day average of the stock price has decreased and sell if the 10-day average has increased. If both conditions are met, then prioritize selling.
- Buy if the 10-day average of the stock price has decreased and sell if the 10-day average of the stock price has increased.
- Randomly buy or sell by flipping a coin, as long as the available cash or the number of stocks remaining allows such an action.
- Buy if the 50-day average of the stock price has decreased and sell if the 50-day average of the stock price has increased.

Rules are ranked according to their strength, initially zero. Every trader has a mental model that learns how good the condition-action rules are based on past experience. If the wealth of a trader has increased after applying a rule, then the strength of this rule increases by one; otherwise, it decreases by one. In each round, traders act according to the strongest rule for the given conditions with a certain probability.

7.1.3 Epidemiology

The classic SIR model takes a top-down approach and describes how an infectious disease spreads in a population via a set of differential equations that model the rate of changes for different states. This makes it difficult to evaluate the aggregated impact of (1) behavioral changes at the level of individuals, such as introducing quarantine requirements that isolate agents for some time, and (2) spatial changes when agents move and interact with different people. Adapting the equations of a SIR model to include the behavior of other identities, such as hospitals or governments, that in turn changes agents' behavior, is also not straightforward. Agent-based simulations address these issues by taking the bottom-up approach, modeling how the disease progresses in a population as people interact, mirroring a real-world scenario.

We extend the classic SIR model with some extra states, E, H, and D. 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 contacting 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.

The transition between different states is probabilistic and modeled after the statistics of COVID-19 (Imperial College COVID-19 Response Team, 2020). The likelihood of falling severely ill is subject to how vulnerable a person is. For simplicity, we assume that the vulnerability level depends only on age. Senior people who are infected are more likely to become hospitalized than young people.

The connectivity of the population 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 – the likelihood of making a susceptible person transition to state I when being contacted – is randomly generated from a gamma distribution with mean 1 and a shape parameter 0.25 (Imperial College COVID-19 Response Team, 2020).

We consider the following random graph models to generate our social network graph.

- 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 (Bollobás, 2001). All edges are independent.
- The stochastic block model (Holland et al., 1983) (SBM) 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.

7.2 Benchmark Evaluation

We have described from a high level how the aforementioned benchmark examples can be used in their respective research fields. Here we make it more concrete, demonstrating some application-level results after running these examples. Specifically, we explain how these examples can be tuned to achieve different outcomes. This is particularly useful if users want to use agent-based simulations as a learning model, such as reverse-engineering how a macro-phenomena can be built from micro-interactions, by adjusting different parameters in order to achieve a better approximation of an expected result.

We emphasize that the application-level semantics of these benchmark examples, which we

```
mean = 0.8

// Uniform distribution
if (Random.nextDouble() < mean) {
  new Cell(true)
} else {
  new Cell(false)
}

// Gaussian distribution
prob = mean + Random.nextGaussian()
if (Random.nextDouble() < prob) {
  new Cell(true)
} else {
  new Cell(false)
}
```

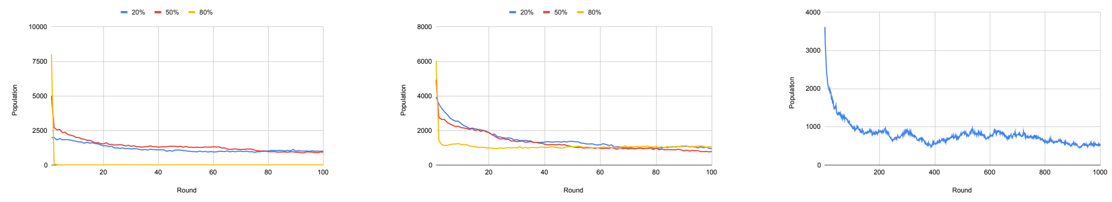
Figure 7.1: Pseudocode that shows different ways of initializing agents in population dynamics simulation.

will describe shortly, mainly concerns users who design similar simulations. Our goal in this section is *not* to find the best configuration of each benchmark example such that it best reflects some existing data set. Instead, we demonstrate *how* different parameters can affect the outcome of a simulation example at the application level, leaving it to users to fine-tune different factors if desired.

7.2.1 Population Dynamics

In population dynamics, we can tune the percentage of agents that are initially alive and how they are distributed, as shown in the pseudocode in Figure 7.1. Figure 7.2 demonstrates how the size of the population changes as the simulation proceeds. We consider two underlying distributions: uniform and Gaussian distribution with mean x (where x is the percentage of initial alive agents) and standard deviation 1.

Depending on the state of their neighbors, agents can die of overpopulation or underpopulation. When initially there are 80% of alive agents, Figure 7.2 shows that the population size drops sharply at the beginning of a simulation due to overpopulation. However, while the population reduces to nearly 0 in (a), the size of the population remains at around 1000 in (b). After some time, it is possible for a population to enter a steady state, where agents are neither surrounded by too many nor too few neighbors. But in general, the population changes dynamically, as illustrated in (c) over a longer period of time (1000 rounds), for a configuration in (b) where 20% of agents are initially alive.



(a) Uniform (b) Gaussian (c) Dynamics of population

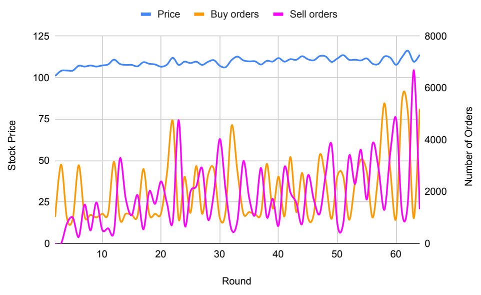
Figure 7.2: Initially, each cell in the population dynamics example has a probability x (20%, shown in blue; 50%, shown in red; and 80%, shown in yellow) of containing an alive agent, drawing from (a) uniform distribution, and (b) Gaussian distribution with mean x and standard deviation 1. Population changes dynamically as a simulation proceeds, which is better illustrated in (c) over a longer period (1000 rounds).

7.2.2 Economics

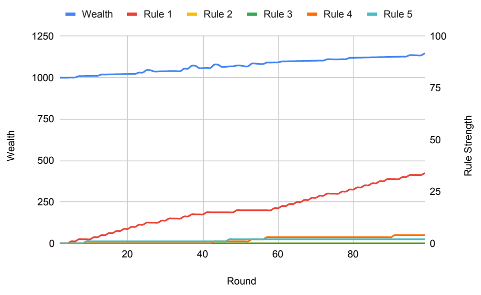
For an evolutionary stock market, the stock price can be viewed as an emergent property of the buy and sell actions of traders. Figure 7.3 (a) demonstrates this process in our benchmark example, where the stock price increases as the number of buy orders increases, and decreases as the number of sell orders increases.

Traders are rule-based learning systems that decide what action to take based on the state of the market and past experience, with the goal of maximizing their wealth. Figure 7.3 assumes that 70% of the time, traders select the rule with the highest strength, otherwise a random rule is selected. Subfigures (b) and (c) show how the strength of different rules changes in two traders as a simulation proceeds, as well as how traders' estimated wealth changes. We notice that in general, the strength of each rule is highly skewed for a given trader. However, this does not imply that a rule with low strength in a particular trader is not efficient in general, as this rule can have the highest strength in other traders. For instance, a trader in (b) learns that rule 1 works the best. Within the same simulation, another trader in (c) learns that rule 3 yields the best outcome instead.

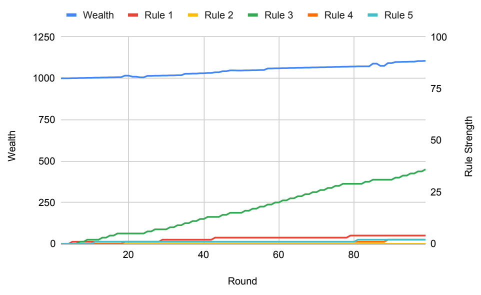
This skew in the strength of learned rules in a trader is partially due to the initial effect. Once a rule has gained a heads-start compared with other rules, even though this rule may have been randomly selected, then it becomes the rule with the maximum strength, hence its probability of being selected again (70% in this example, though tunable) is much higher than other rules, which contributes to the skew. While there are ways to address it, this skew is not necessarily a concern. To illustrate, consider the situation where traders always select a rule at random, shown in (d). Although the gap between the strength of different rules has now been greatly reduced, the wealth of the trader in (d) is lower than that of (b) and (c).



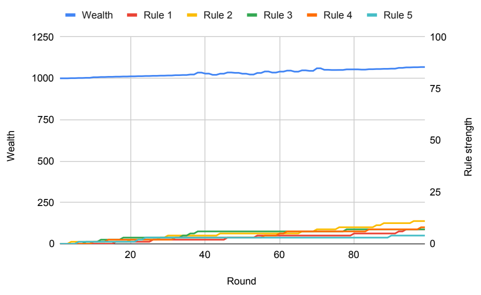
(a) Stock price vs trader actions



(b) Rule 1 works best

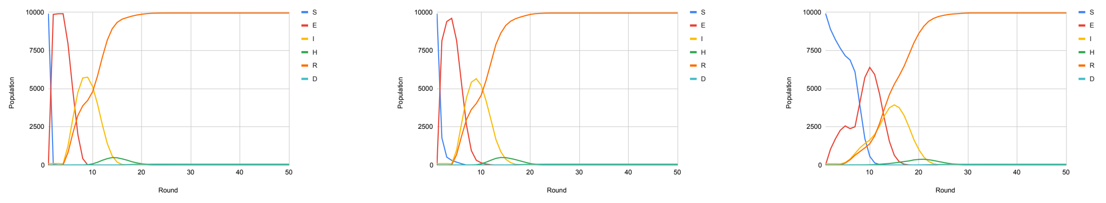


(c) Rule 3 works best



(d) Random rules

Figure 7.3: In an evolutionary stock market, (a) the stock price (blue) is regarded as an emergent property of the buy (orange) and sell (pink) actions of traders. As a simulation proceeds, traders learn different rules. In (b) and (c), traders select the rule with the highest strength with a probability of 0.7. The strength of each rule is highly skewed for a given trader. However, this does not necessarily imply that a rule with low strength in a particular trader is not efficient in general, as this rule can have the highest strength in other traders. For (b), rule 1 (red) has the highest strength. Within the same simulation, (c) learns that rule 3 (green) works best instead, though both traders have similar initial setups. This skew can be addressed by assuming each rule is selected at random with the same probability, as shown in (d). However, the wealth (blue) of traders in (b) and (c) are higher than that of (d).



(a) p=0.3 (b) p=0.1 (c) p=0.01

Figure 7.4: Tune the edge probability p to modify the topology of an ERM graph. As p decreases (a) 0.3, (b) 0.1, and (c) 0.01, the blue curve (S) shifts towards the right and the maximum number of exposed (E, red curve) or infectious (I, yellow curve) population per round lowers.

7.2.3 Epidemiology

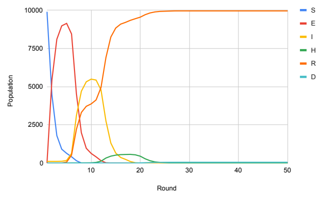
Compared with previous examples, the epidemics examples include much more parameters and offer an extremely high degree of flexibility for customization, including connectivity of the underlying social graph, duration of different states, probability of state transitions, and population distribution, which we will explore here. We emphasize that the aforementioned factors are far from an exhaustive list. For instance, users can also adjust the vulnerability level of different people and the probability of various state transitions.

Tune the connectivity of social network graphs. We start by adjusting the topology of the underlying random graph through parameters such as edge probability, as explained before (see Section 7.1). When the edge probability decreases, the average number of neighbors per agent also decreases. Consequently, the number of people being contacted by infectious individuals in a round is also reduced, hence the spread of disease should slow down.

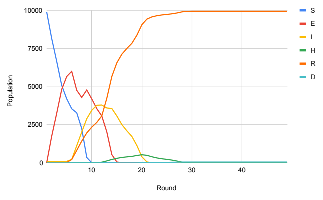
Figure 7.4 demonstrates that as edge probability p decreases in ERM, the disease transmission indeed slows down: the curve that corresponds to S (blue) is shifted towards the right, hence the number of rounds for the entire population to become exposed to the disease has increased. Consequently, other curves have also been shifted accordingly with a lower peak value. For instance, the maximum number of exposed (E, red curve) or infectious (I, yellow curve) people in any round becomes lower.

The analysis for SBM experiments shown in Figure 7.5 is similar. The disease transmission slows down as (a) within-block edge probability p decreases, (b) cross-block edge probability q decreases, and (c) the number of blocks n increases.

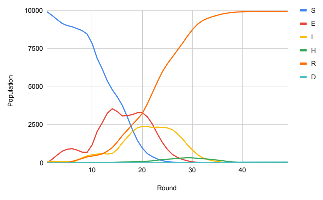
Tune the number of contacts. By default, we assume that an infectious agent contacts all its neighbors in each round. We can also limit the maximum number of contacts that an agent can make per round. If we decrease the maximum number of contacts per agent from 50 agents to 10 agents, we see in Figure 7.6 that the number of rounds that it takes for the susceptible population to become exposed has increased by several factors (blue curve has



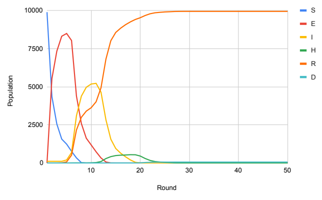
(a) p=0.3, q=0.001, n=5



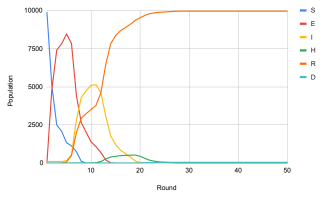
(b) p=0.1, q=0.001, n=5



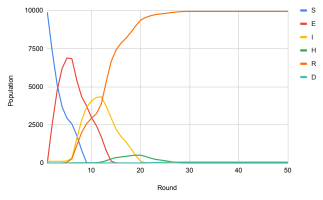
(c) p=0.01, q=0.001, n=5



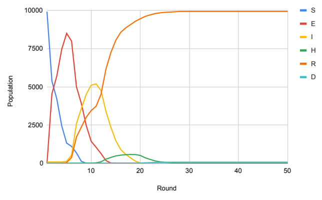
(d) p=0.1, q=0.1, n=5



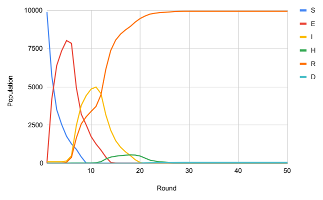
(e) p=0.1, q=0.05, n=5



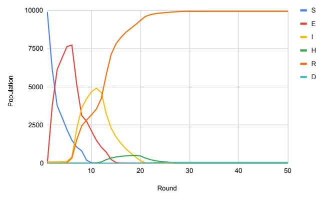
(f) p=0.1, q=0.01, n=5



(g) p=0.1, q=0.01, n=10

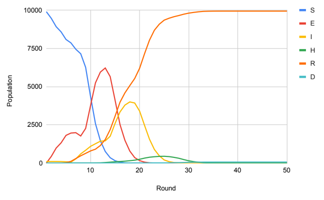


(h) p=0.1, q=0.01, n=20

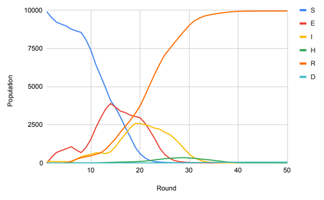


(i) p=0.1, q=0.01, n=50

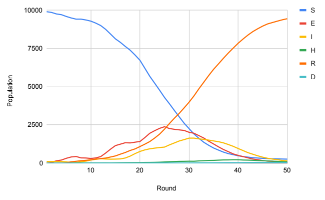
Figure 7.5: Tune the edge probability within-block (p), between-block (q), and the number of blocks n, to modify the topology of a SBM graph. We first assume q=0.001 and n=5 and decrease p in (a) 0.3, (b) 0.1, and (c) 0.01. Subsequently, we fix p=0.1 and n=5, lowering the value of q in (d) 0.1, (e) 0.05, and (f) 0.01. Finally, we set p=0.1 and q=0.01, raising the number of blocks n in (g) 10, (h) 20, and (i) 50. In all such cases, i.e. when either p or q is decreased or when n is increased, we observe a notable reduction in disease circulation.



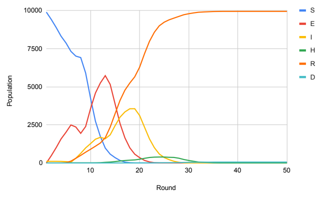
(a) ERM, 50 contacts



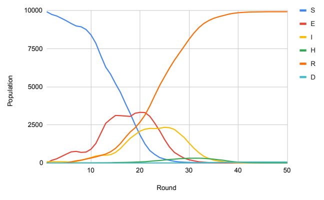
(b) ERM, 20 contacts



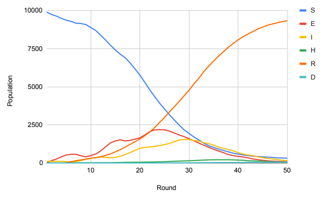
(c) ERM, 10 contacts



(d) SBM, 50 contacts



(e) SBM, 20 contacts



(f) SBM, 10 contacts

Figure 7.6: Curtail the maximum number of people that an infectious individual can contact in ERM (p=0.1) (a-c) and SBM (p=0.1, q=0.01, n=5) (d-f), from 50 (a, d) to 20 (b, e) and 10 (c, f) can mitigate the spread of disease.

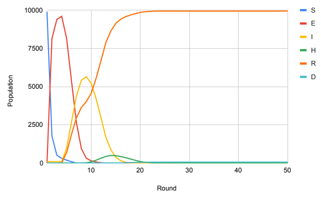
shifted to the right), hence this is effective at curbing the disease transmission.

Reducing the maximum number of contacts that infectious agents can make is also the core logic behind non-pharmaceutical interventions (NPIs), including quarantine or lockdown. The effectiveness of different NPI strategies can be understood from how such strategies limit the number of contacts per agent.

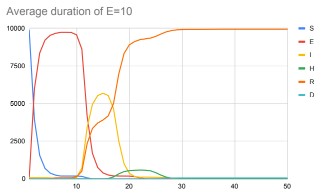
Tune state durations. So far we have discussed factors such as adjusting the topology of a social graph or limiting the maximum number of contacts per agent, which address how fast a disease spreads by targeting *interactions* between agents. In the experiments here and below, we analyze other factors that are independent of how people interact, such as the course of the disease in a person, including the duration of different stages of the disease – which manifests into people being in different health states (SEIHRD) – and the likelihood of transitioning into other stages.

We first tune the average duration of different states, using E and I as examples. Per state, its duration is a high-level description of how many rounds an individual remains in this state. Not all states have such a duration, such as S, R, and D. People remain R or D until a simulation ends. For S, healthy people stay in S up to being infected by infectious people. In particular, the duration does not concern *transitions* that are associated with this state. For instance, users may assume that on average agents remain infectious (I) for 5 rounds. This does not make any presumption on the state that an agent transits to afterward, such as R, H, or D.

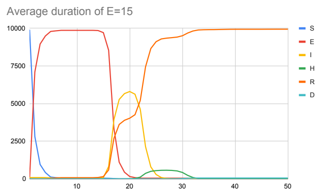
Figures 7.7 and 7.8 show that as we increase the average duration of E or I from 5 to 10 and 15 rounds, curves that correspond to these states have been widened in such a way that the



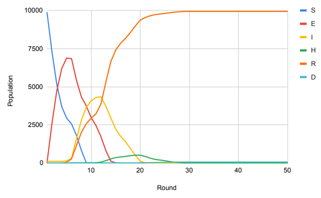
(a) ERM, E duration 5



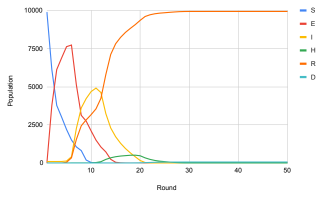
(b) ERM, E duration 10



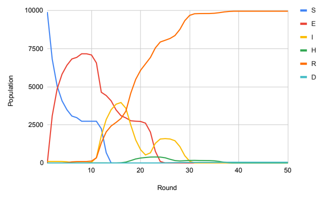
(c) ERM, E duration 15



(d) SBM, E duration 5



(e) SBM, E duration 10



(f) SBM, E duration 15

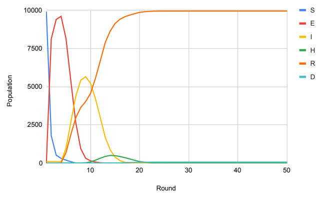
Figure 7.7: Increase the average duration of E in ERM (p=0.1) (a-c) and SBM (p=0.1, q=0.01, n=5) (d-f), from 5 (a, d) to 10 (b, e) and 15 (c, f). Curves that correspond to E (red) have changed in such a way that the width of their respective peaks has been expanded to reflect the increase in the average duration.

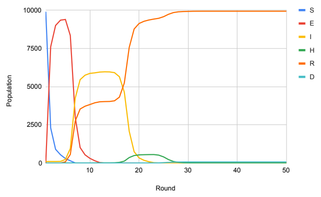
"width" of their respective peaks has increased to reflect the new average duration. To clarify, let m_e denote the maximum number of people in E in a round when the average duration is 5. After increasing the average duration to 10 or 15, although the maximum number of people in E in a given round does not change much from m_e, the number of rounds that a total of m_e people remain in E has been increased to approximately the new average duration.

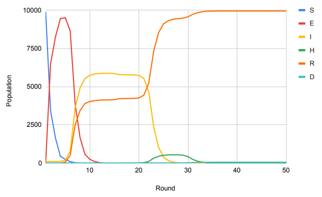
Tune state transition probabilities. Here we quantify the impact of transition probability on the spread of disease, using the probability of low-risk individuals transitioning from state E to I and from I to H as examples.

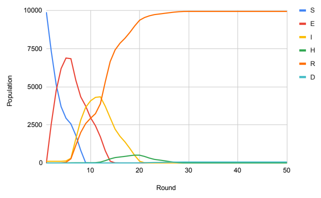
In previous experiments, the probability of low-risk people transitioning from E and I is 0.6. During this experiment, we decrease this probability to 0.3 and 0.1 respectively. Figure 7.9 shows that curves that correspond to I have been lowered such that the "height" of their peak, measured in the size of the population, decreases to approximately 1/2 and 1/6 of the original height (when the probability is 0.6), to reflect the new transition probability. Curves for other states are changed accordingly. The same observation is made when decreasing the probability of low-risk people transitioning from I to H from 0.1 to 0.05 and 0.01 respectively. Figure 7.10 shows that the curve corresponding to H has been similarly lowered.

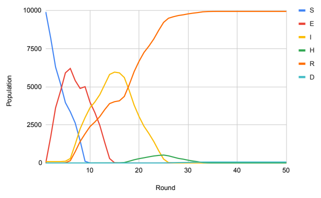
Tune demographic profile. We also evaluate how demographic profile changes the spread of disease. In particular, we consider the impact of age distribution. Our example models the age distribution using Gaussian with a standard deviation of 1 and a mean that reflects average age. Previously we assumed the average age to be 45, and here we adjust it to 25 and


(a) ERM, I duration 5


(b) ERM, I duration 10


(c) ERM, I duration 15


(d) SBM, I duration 5


(e) SBM, I duration 10

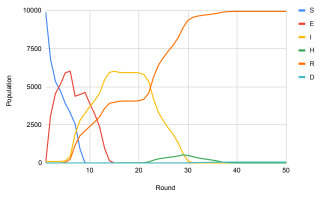
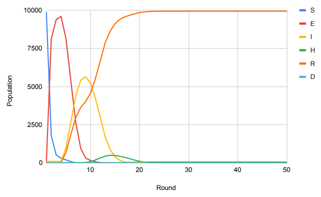
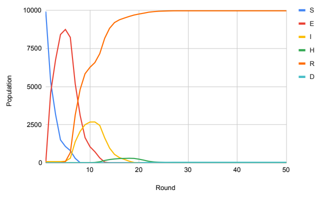
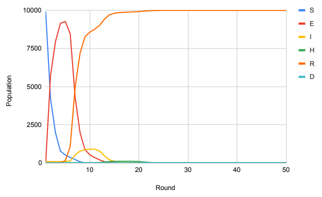
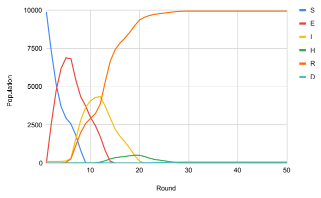

(f) SBM, I duration 15

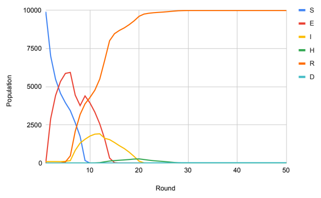
Figure 7.8: Increase the average duration of I in ERM (p=0.1) (a-c) and SBM (p=0.1, q=0.01, n=5) (d-f), from 5 (a, d, also presented in figures 7.4 and 7.5) to 10 (b, e), and 15 (c, f). Curves that correspond to I (yellow) have been widened such that the width of their respective peaks has increased to reflect the new average duration, similar to Figure 7.7.


(a) ERM, P_{E2I}=0.6


(b) ERM, P_{E2I}=0.3


(c) ERM, P_{E2I}=0.1


(d) SBM, P_{E2I}=0.6


(e) SBM, P_{E2I}=0.3

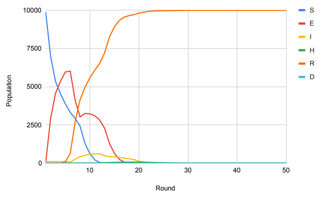
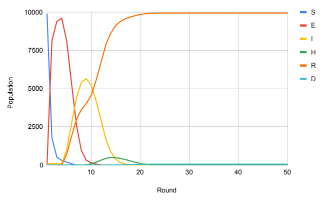
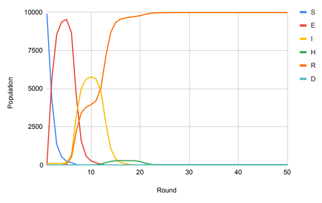

(f) SBM, P_{E2I}=0.1

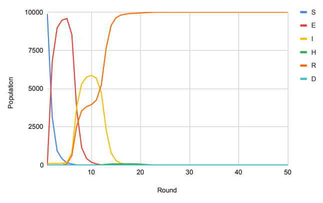
Figure 7.9: Decrease the probability of transitioning from state E to I for low-risk people in ERM (p=0.1) (a-c) and SBM (p=0.1, q=0.01, n=5) (d-f), from (a, d) 0.6 to (b, e) 0.3 and (c, f) 0.1.



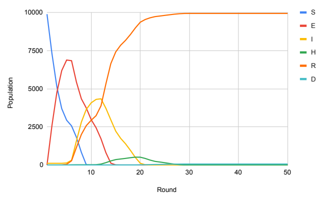
(a) ERM, P_{I2H}=0.1



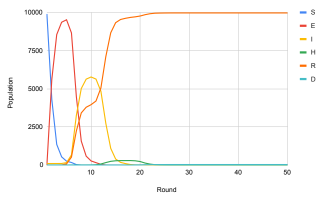
(b) ERM, P_{I2H}=0.05



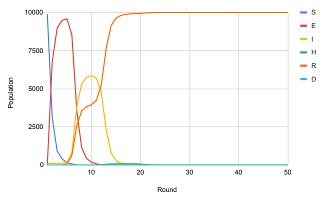
(c) ERM, P_{I2H}=0.01



(d) SBM, P_{I2H}=0.1



(e) SBM, P_{I2H}=0.05



(f) SBM, P_{I2H}=0.01

Figure 7.10: Decrease the probability of transitioning from state I to H for low-risk people in ERM (p=0.1) (a-c) and SBM (p=0.1, q=0.01, n=5) (d-f), from (a, d) 0.1 to (b, e) 0.05 and (c, f) 0.01.

65 respectively.

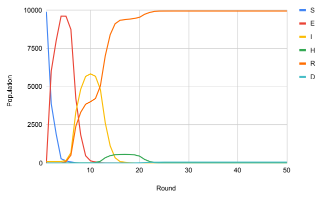
As the average age increases, the population of high-risk individuals also increases. Figure 7.11 shows that when the average age is 65, the population of H and D have increased significantly compared with when the average age is 45, but the change is less dramatic when increasing the average age from 25 to 45. This is because we consider people above 60 to be high-risk, which can also be customized.

7.3 Implementation Details

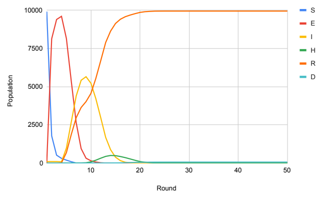
We have shown how users can adjust different application-level parameters to fine-tune the outcome of a simulation example in our benchmark. For our purpose, however, we are interested in the low-level characteristics of these applications, such as their communication pattern and computation pattern, and use these workloads to benchmark the relative performance of different BSP-like systems when executing agent-based simulations.

We implement the workloads such that the agent code in CloudCity or other systems has the exact same behavior.[1] 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 can not 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

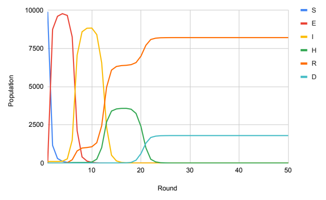
[1]This is modulo potential non-determinism from randomized computation or the semantics of consuming messages from the mailbox in an arbitrary order.



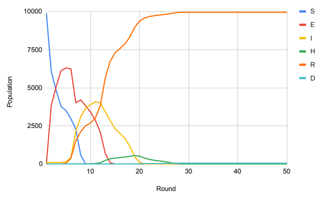
(a) ERM, average age 25



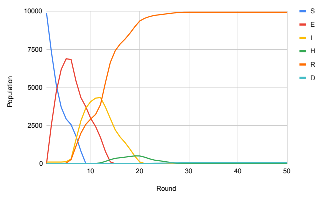
(b) ERM, average age 45



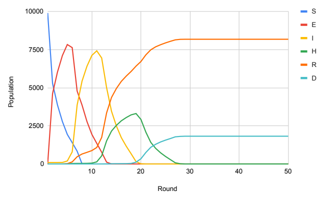
(c) ERM, average age 65



(d) SBM, average age 25



(e) SBM, average age 45



(f) SBM, average age 65

Figure 7.11: Tune the average age of the population in ERM (*p*=0.1) (a-c) and SBM (*p*=0.1, *q*=0.01, *n*=5) (d-f), from (a, d) 25 to (b, e) 45 and (c, f) 65.

accesses and deforestation that depend on such functionalities are inapplicable in CloudCity.

The programming model of Spark is RDD-based, where an RDD (Resilient Distributed Dataset) is an immutable collection of in-memory objects that can be parallelized (Zaharia et al., 2012). This allows Spark to achieve better performance than disk-based systems, such as Hadoop (Apache Hadoop Developers, 2006). Agents in CloudCity, Giraph, and Flink Gelly are also stored in memory. We consider two possible implementations in Spark, stateless and stateful. In the stateless approach, the "state" of an agent is captured in a data structure and is updated as a simulation progresses through the following APIs: the `run` method consumes this data structure and a collection of incoming messages, and returns the updated agent state after processing the messages; the `sendMessage` method consumes the state of an agent and returns one or more messages generated by the agent that will be sent to all connected agents. For instance, in the population dynamics example, the signature of `run` takes a Boolean attribute that denotes whether an agent is alive as input and returns it as output:

```
run(id:Long,alive:Boolean,messages:List[List[Boolean]]):Boolean;
```

similarly for epidemics example, where `run` takes a list of integers that represents the state of a person, including age, health status, and vulnerability, as an input and returns it as output:

```
run(id:Long,state:List[Int],messages:List[List[Double]]):List[Int].
```

It is easy to see that the entire state of a vertex is captured in the input and output data flow of `run`, hence "stateless". This contrasts with a stateful implementation, where

```
run():Unit
```

is defined inside a stateful agent object and mutates the state of an agent like

```
class Vertex extends Agent {
  var alive: Boolean = Random.nextBoolean()

  def run(): Unit = {
    // do sth
    ...
  }
}
```

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 (Zaharia et al., 2012). 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 thesis) showed that changing agents to stateless is orders of magnitude faster due to reduced object creation and copying in Spark.

We adopt stateless agents for the Spark implementation when compare with other systems. The 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.

Giraph is an open-source implementation of Pregel (Malewicz et al., 2010), a large-scale graph processing system developed by Google. Users define a graph algorithm in Giraph as a vertex program with a `compute` method, which is executed iteratively by all vertices in an input graph, as described in the BSP model. In every superstep, each vertex in the input graph receives messages sent from neighbors in the previous superstep, executes the `compute` method, and sends messages to other vertices. GraphX is a graph library embedded in Spark and introduces optimized graph-specific operators. Gelly is a graph library of Flink (Developers, n.d.), which provides a unified programming model for both batch and stream processing (Carbone et al., 2015). 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.

7.3.1 System Comparison

Table 7.1 summarizes system features that 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, Giraph, Flink Gelly, Spark, and GraphX for simulations, by analyzing the applicability of the features listed in Table 7.1 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 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 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.

Table 7.1: Features of different systems

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

7.4 System Evaluation

We examine the relative performance of CloudCity compared to other state-of-the-art BSP-like systems using 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 different 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 Chapter 6 and the performance impact of hierarchical communication patterns with delayed messages separately.

We make default application-dependent parameters that are used in the subsequent experiments explicit for each workload. For the population dynamics example, half of the cells are initially alive. The population distribution forms a uniform distribution. In the economics example, we consider one market agent and tune the number of trader agents. For simplicity, we assume traders always select the condition-action rule that has the highest strength. Regarding the epidemics examples, the social network graph is generated by ERM ($p = 0.01$) and SBM (5 blocks, $p = 0.01$, $q = 0$). We use SBM just to have a clear block structure and set $q = 0$. Per round, an infected agent sends a message to all its connected neighbors. The average state duration of E, I, and H is 5, 5, and 7 respectively. The state transition probability for low-risk people from E to I, I to H, and H to D is 0.6, 0.1, and 0.1 respectively. For high-risk people, the corresponding transition probability is 0.9, 0.4, and 0.5. The average age of the population is 45, and the age profile is generated according to a Gaussian distribution with a mean of 45 and a standard deviation of 1.

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 example, 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.

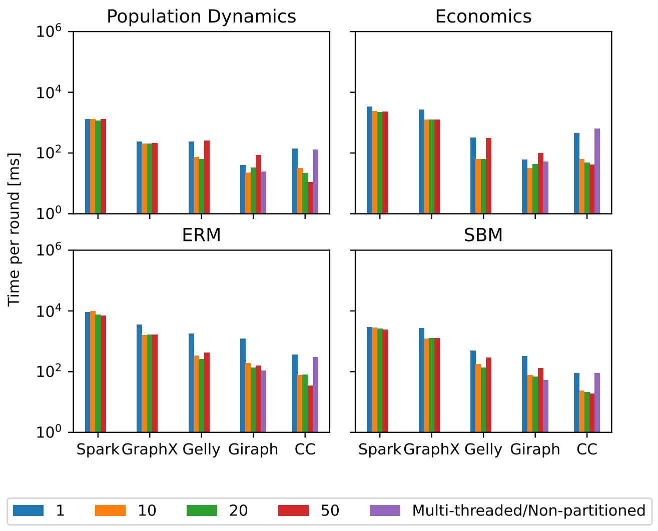


Figure 7.12: Finding the best configuration of each system for 10,000 agents on one machine (unit of parallelism is system-specific)

7.4.1 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 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 7.12. 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 (Apache Spark Developers, 2018). Figure 7.12 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 7.12, 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 7.12 shows that one multi-threaded worker per machine has the best overall performance across different workloads, as also reported in previous work (Ching et al., 2015).

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 7.12 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.

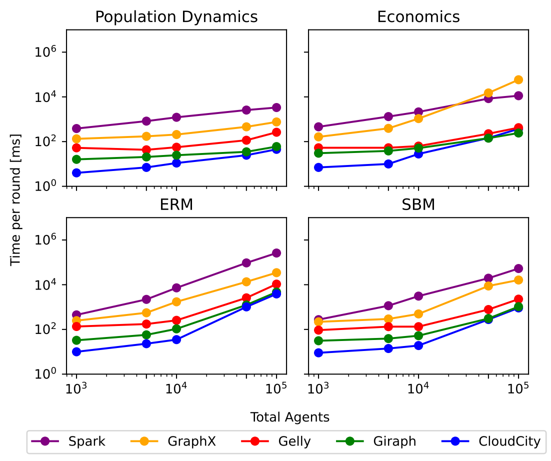
Figure 7.12 shows that the partitioned execution with 50 components is 4-15× faster than the non-partitioned execution, even though all agents in CloudCity are configured to be 1-available (Section 7.3) and synchronize every round in both executions. There are several reasons for the speedup. Firstly, 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.

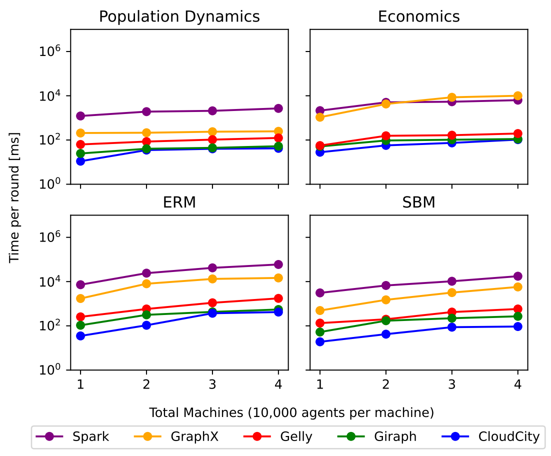
7.4.2 Scalability

In this experiment, we compare the scalability of these systems by increasing the number of agents (Figure 7.13a) from 1,000 to 100,000, and the number of machines from 1 to 4 (Figure 7.13b). 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



(a) Scalability (scale-up)



(b) Scalability (scale-out)

Figure 7.13: Scalability experiments (a) Increasing the number of agents; (b) Increasing the number of machines.

Giraph. In the SBM experiment (Figure 7.13b 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 7.1. 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 (Gonzalez et al., 2014) 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.

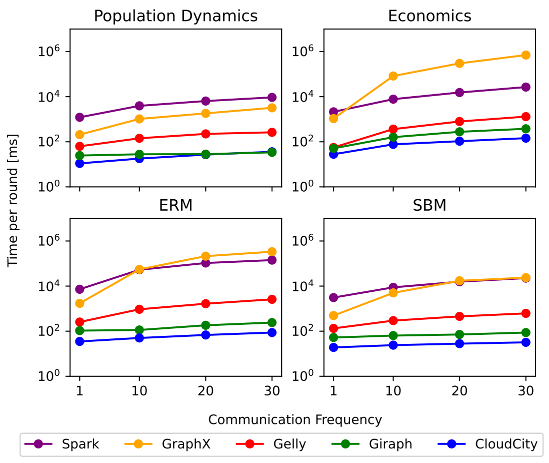


Figure 7.14: Microbenchmark: Increasing communication frequency

7.4.3 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 7.14 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.

7.4.4 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 intermittent computation pattern, such that agents only communicate every n rounds, where $n = 1$ is the base case (no change from before), shown in Figure 7.15. 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 7.3. The lack of support for expressing idle periods in such systems also has a performance cost. For instance,

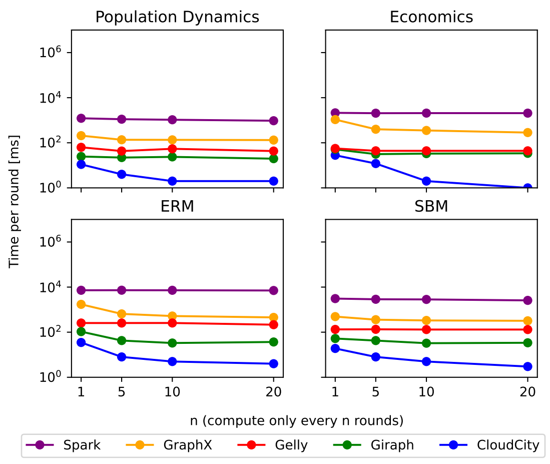


Figure 7.15: Microbenchmark: Increasing the number of rounds until the computation is resumed

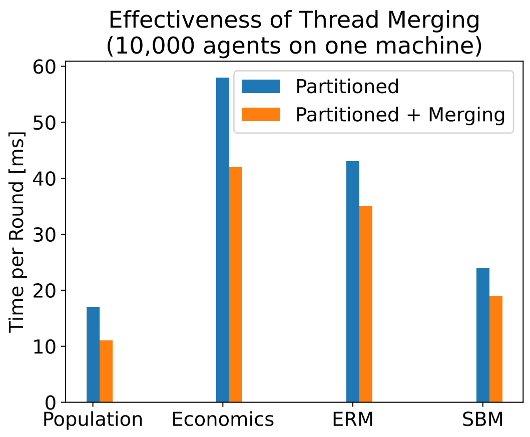
CloudCity is 10× faster than Giraph when $n = 20$, but the performance of these two systems is on par when $n = 1$.

7.4.5 CloudCity-Specific Optimizations

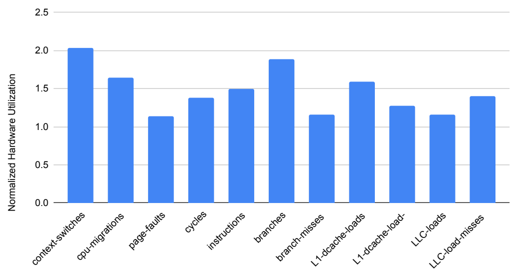
From now on, we assess system optimizations available only in CloudCity. For experiments in this section, we only evaluate CloudCity on one machine with 10,000 agents, unless specified otherwise.

Thread Merging. We evaluate the effectiveness of thread merging across all workloads. Same as before, we separate agents evenly into 50 components. Figure 7.16a 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 improves the hardware utilization rate by lowering the number of context switches and the number of branches, as illustrated in Figure 7.16b.

Fusing Send and Receive Operations. We evaluate the effectiveness of fusing send and receive operations 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.



(a) Effectiveness across all workloads



(b) The ratio of hardware performance counter values before and after thread merging for the economics example

Figure 7.16: Thread merging (a) improves the performance of all workloads by up to 2×. This performance benefit is from better scheduling and more efficient hardware utilization. For instance, in (b) the economics example, the number of context switches and the number of branches is reduced by half after thread merging.

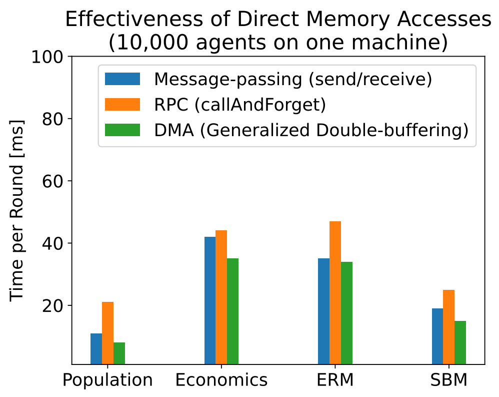
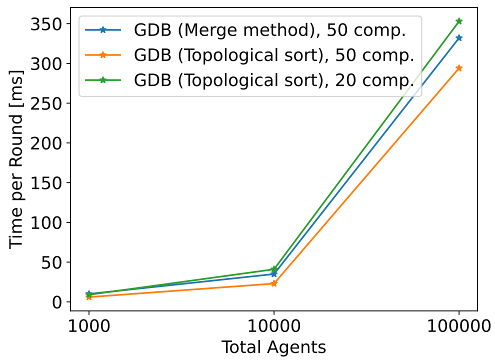
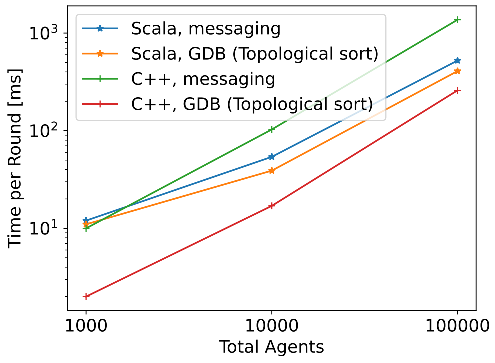


Figure 7.17: Fusing send and receive operations using generalized double-buffering (GDB) with an additional merge method improves the performance by up to 2× among all workloads.



(a) GDB with topological sort (y-axis is linear)



(b) C++ vs Scala (y-axis is log-scale)

Figure 7.18: In the case of economics example, (a) exploiting topological sorts of the DMA graph (orange) to avoid applying the merge method improves the performance further for 50 components. There is a tradeoff between parallelizing computation and transforming more messages to DMAs. As we reduce the number of components from 50 to 20 (green), more messages can be transformed to DMA, but less computation can be parallelized. Here 20 components has worse performance than 50 components. (b) Using a low-level language with fine-grained memory management like C++ can obtain much higher improvement. When merging all agents into a single component, fusing send and receive operations in C++ (red) is 6× faster than messaging (green) and 2× faster than fusing in Scala (orange).

We experiment with both multi-versioning implementations and generalized double-buffering implementations. The performance of multi-versioning is 2-3× worse than generalized double-buffering for all examples, hence not showing here. Such performance differences is due to memory activities for a JVM-based language like Scala. While generalized double-buffering pre-allocates memory needed for the auxiliary attributes, multi-versioning 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.

Here we focus on generalized double-buffering, and begin with generalized double-buffering that has an additional merge method. In particular, we define RPCs in such a way that received inputs are simply buffered, without applying any partial evaluation to compute the results.

To clarify, we use the economics simulation as an example and show how the market agent definition is transformed from message-passing

```
class Market extends Actor {
  // ... states
  def main(): Unit = {
    while (true) {
      buyOrders = 0
      sellOrders = 0

      var m = receiveMessage()
      while (m.isDefined){
        var ans = m.get.value
        if (ans == 1) {
          buyOrders = buyOrders + 1
        }
        if (ans == 2) {
          sellOrders = sellOrders + 1
        }
        m = receiveMessage()
      }

      // ... other computations
      // send messages to neighbors
      connectedAgentIds.foreach(i => {
        // ... construct message
        sendMessage(i, msg)
      })
      waitRounds(1)
    }
  }
}
```

to allow generalized double-buffering with an additional merge method

```
class Market extends Actor {
  // ... states
  @allowDirectAccess
  def traderAction(action: Int): Unit = {
    bufferedActions.append(action)
  }

  // The merge method is called at the beginning of each round,
  // before any agent starts executing
  def merge(): Unit = {
    buyOrders = bufferedActions.count(_ == 1)
    sellOrders = bufferedActions.count(_ == 2)
    bufferedActions.clear()
  }

  def main(): Unit = {
    while (true) {
      handleRPC()
      // notify traders of the latest market state
      traders.foreach(x => callAndForget(x.inform(stockPrice,
  marketState), 1))
      wait(1)
    }
  }
}
```

Notice that the RPC method `traderAction` simply buffers the received trader actions.

For each example in the benchmark, we apply such transformations to allow agents to use generalized double-buffering with an additional merge method. Figure 7.17 shows the evaluation of this optimization compared with message-passing. Since RPC methods are simply buffering user inputs, it is unsurprising that this general transformation does not lead to significant speedup, which is up to 2×. 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 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 consider a special scenario where agents use generalized double-buffering but without a merge method, by exploiting a topological sort of a DMA graph of a simulation, as we have explained previously in Section 6.2. We use the economics example as a case study. In this experiment, we avoid the overhead of buffering user inputs and change the market agent definition to allow partial computation in the RPC method, shown below. Agents are still parallelized and separated into 50 components. In the component that contains the market agent, the market agent is merged before any traders. Messages sent from traders in the same component are compiled away and transformed into DMAs. Traders in other components still communicate with the market agent by messaging.

Figure 7.18a shows that for 50 components, exploiting topological sort (orange) achieves slightly better performance than applying the merge method (blue), but not by much. We point out that there is a tradeoff when determining the number of components before applying DMA. On the one hand, increasing the number of components reduces the average number of agents per component, hence the performance benefit from exploiting the within-component communication is more limited compared with merging all agents into a single component. On the other hand, increasing the number of components reduces the amount of time needed to execute all agents within the component, hence can lead to better parallelization and faster overall execution time. For the economics example, increasing the number of agents per component, by reducing the number of components from 50 (orange) to 20 (green), hurts the performance instead.

```
class Market extends Actor {
  // ... states
  @allowDirectAccess
  def traderAction(action: Int): Unit = {
    if (action == 1){
      buyOrders += 1
    } else if (action == 2) {
      sellOrders += 1
    }
  }

  def main(): Unit = {
    while (true) {
        handleRPC()
        stockPrice = stockPriceUpdate(buyOrders, sellOrders)
        // notify traders of the stock price and latest market state
        traders.foreach(x => callAndForget(x.inform(stockPrice,
    marketState), 1))
        wait(1)
      }
    }
  }
```

Though our empirical evaluation of fusing send and receive operations does not achieve orders of magnitude speedup, we do believe that this is a promising optimization that has more potential. However, taping this optimization in a high-level language like Scala whose performance is easily affected by memory activities, as evident from the performance gap between multi-versioning implementation and generalized double-buffering, can be difficult. Using a low-level language like C/C++ that offers fine-grained control over memory activities can better exploit this optimization.

To demonstrate, we implement a C++ prototype of a non-distributed backend for DMA experiments and port the economics example, merging all agents into one component. Figure 7.18b demonstrates that in the C++ implementation, DMA with topological sort (red) is 6× faster than messaging (green), consistently as we increase the number of agents from 1,000 up to 100,000. Fusing send and receive operations in C++ is also 2× faster than that in Scala (orange). We also notice that in C++, messaging is up to 3× slower than that in Scala (blue), due to just-in-time compilation and other Java Virtual Machine (JVM) run-time optimizations when executing Scala.

Tiling. Tiling separates the social graph of a simulation into tile agents and only sends messages between these tile agents, as described in Section 6.3. We consider the population dynamics example, which is embedded in a 2D grid. More concretely, we present the pseudocode of a tile agent for this example below.

```
class TileAgent extends Agent {
  // ... states and computations

  def step(): Int = {
    // ... computation to process messages
    for (i <- (1 to rows-1)) {
      for (j <- (0 to cols-1)) {
        newBoard(i)(j) = actionPerVertex(Coordinate2D(i, j))
      }
    }
    oldBoard = newBoard
    1
  }
}
```

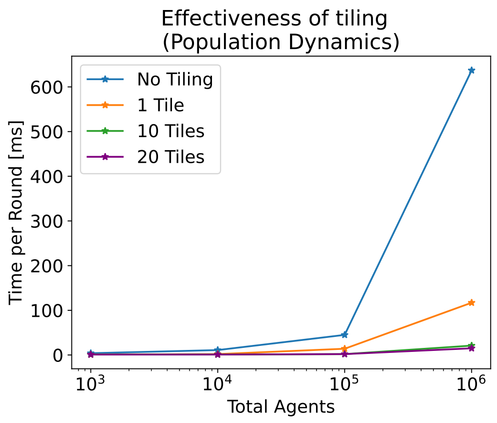


Figure 7.19: Microbenchmark the effectiveness of tiling in the population dynamics example. We increase the number of agents from 1,000 to 1,000,000 and raise the number of tiles from 1 to 20. For 1,000,000 agents, using 10 or 20 tiles improves the performance by over 20× compared with no tiling.

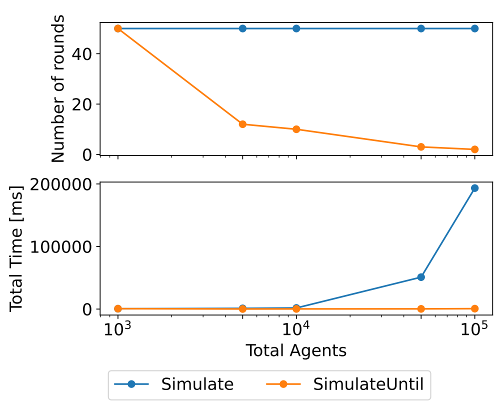
Figure 7.19 shows the performance of tiling compared with message-passing (with label "no tiling"), when increasing the number of agents from 1,000 to 1,000,000 agents, and increasing the number of tiles from 1 to 20. We see that using 10 or 20 tiles improves the performance by over 20× for 1,000,000 agents compared with no tiling.

`SimulateUntil` **Operator.** Recall that `SimulateUntil` allows users to stop a simulation early. It is clear that the speedup of `SimulateUntil` depends on user queries. Here we revisit a sample question presented before in Section 6.4, "When does half of the population becomes exposed to the disease (state E)?" and evaluate it in our ERM example (*p*=0.01).

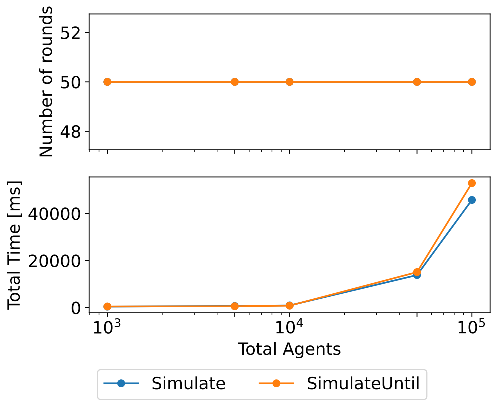
This question can be expressed as a query using `SimulateUntil` like below.

```
// population: Int
SimulateUntil(agents, 50)((ts: Iterable[Iterable[Actor]]) => {
  ts.map(i => {
    i.filter(a => {
      a match {
        case x: epidemic.Person => x.health == Exposed
      }
    }).size > (population / 2)
  })
})
```

The simulation terminates either the condition that at least half of the population becomes exposed evaluates to true or 50 rounds have passed. The performance metric that we consider is *total time* until a simulation completes, which is different from *average time per round* that we have presented so far.



(a) ERM



(b) SBM

Figure 7.20: Microbenchmark the effectiveness of `SimulateUntil` in the context of evaluating user query "When does at least half of the population become exposed (state E) to the disease?" in ERM and SBM respectively, as we increase the number of agents from 1,000 to 100,000. (a) In ERM, the number of rounds that it takes to answer this question decreases as the size of the population increases. The speedup of `SimulateUntil` is up to 300×. (b) In SBM, the number of rounds remains the same as that of `Simulate`. `SimulateUntil` is slightly slower than `Simulate` due to the overhead of evaluating against the time series in each round.

Figure 7.20 shows the evaluation result for our microbenchmark experiments that compare the performance of evaluating the aforementioned user query using `SimulateUntil` and `Simulate` in the epidemics example, for both ERM and SBM. In Figure 7.20a, we notice that the number of rounds that takes to answer the user query in ERM decreases as the number of agents increases: For 1000 agents, the simulation runs until 50 rounds have passed; but for 100,000 agents, it takes only 2 rounds for half of the population to become exposed. Intuitively, in the ERM graph with a fixed edge probability, the number of edges connected with a vertex increases as the number of vertices increases. Hence, the number of neighbors per agent increases as the size of the population grows. The speedup of `SimulateUntil` increases to well over two orders of magnitude as the number of agents increases to 100,000. However, in SBM, the average number of neighbors per agent is much lower than that of ERM. Figure 7.20b shows that simulations using `SimulateUntil` still run for 50 rounds, the same as using `Simulate`. Due to the overhead of evaluating the user-defined condition at the end of each round, using `SimulateUntil` is slightly worse than using `Simulate` in Figure 7.20b.

Deforestation. We assess the effectiveness of deforestation using the population dynamics example as a case study and consider multiple queries, listed in Table 7.2. The "time series" experiments save a copy of each agent at the end of every round to generate a time series, which allows users to query the time series arbitrarily many times. In case users are only interested in a single query, then we can apply deforestation to reduce the intermediate data.

Figure 7.21 shows the average time per round when increasing the number of agents from

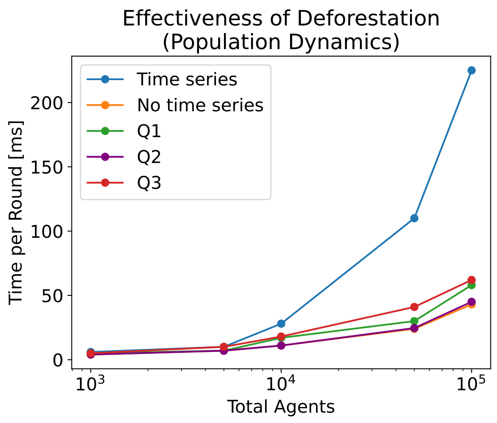


Figure 7.21: Microbenchmark the effectiveness of deforestation in the population dynamics example. We scale up the number of agents from 1,000 to 100,000. The baseline is "time series", which materializes a sequence of collections of agents. For each user query presented in Table 7.2, we measure the average time per round after applying deforestation. Deforestation is effective and eliminates nearly all the overhead of generating a time series, approaching the performance of "no time series".

Table 7.2: 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))`

1,000 to 100,000 while generating the 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 generating the time series.

We emphasize that the speedup of deforestation depends on the selectivity of the user query. For example, if an agent has 20 user-defined attributes and the query concerns only one, then materializing the projected attributes reduces the size of the intermediate data by nearly 20×. Agents in the population dynamics, however, only have one user-defined attribute. Thus, our experiment illustrates the effectiveness of deforestation, even in the worst-case.

Hierarchical Communication. Our system supports hierarchical communication: Agents in the same component communicate faster (messages arrive in fewer rounds) than those in other components.[2] To evaluate the benefit of exploiting this, we configure agents to be K-available to agents in other components every K rounds (see Chapter 3), and 1-available to

[2]In contrast, the communication in Giraph is inherently non-hierarchical.

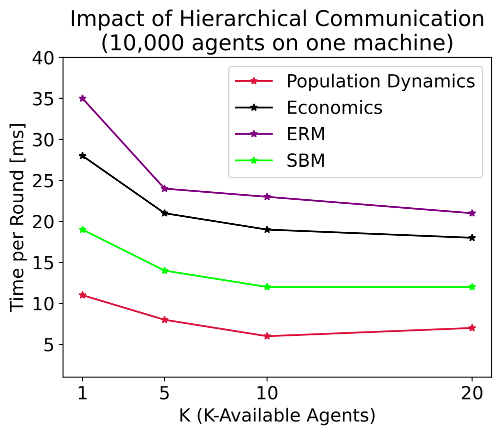


Figure 7.22: Exploiting hierarchical communication with K-availability

those in the same component in each round. We set up agents to be K-available for increasing values of K, from 1 to 20.[3]

Figure 7.22 shows that the performance increases by up to 2× consistently over all workloads. This is a compound impact of improving communication and computation locality for agents in the same component, and a tradeoff between the number of local vs. remote messages. While the performance of most workloads improves as K increases, in the population dynamics, $K = 10$ actually has a better performance than $K = 20$. Because agents send messages to each neighbor in every round, more messages are buffered as K increases. For $K = 20$, the system needs to deliver twice as many messages than for $K = 10$ during merging, thus taking a longer time. If such an increase in the time to merge is less than the time saved from delivering the local messages faster (in terms of wall clock time), then the overall performance increases compared to non-hierarchical communication.

CloudCity++. Finally, we summarize how CloudCity performs relative to other systems in Figure 7.23, with all optimizations enabled (shown with the label CC++), on a single machine with 10,000 agents. Compared with the base performance of CloudCity (shown with the label CC Base), system optimizations improve performance by 2-10×, resulting in 10-20× overall improvement over other systems.

[3]For some workloads, this requires adjusting the semantics accordingly. 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.

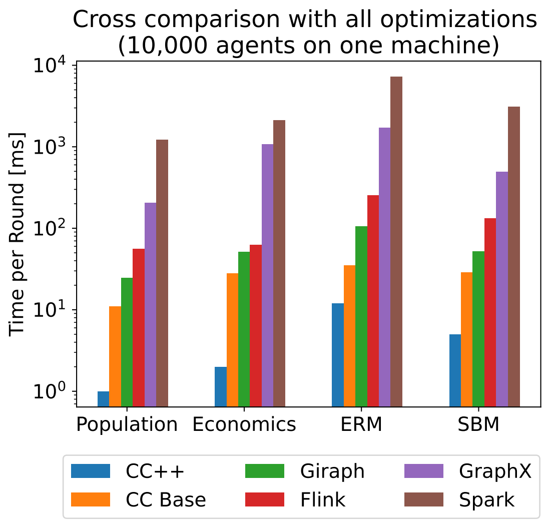


Figure 7.23: Compare CloudCity with other systems (10,000 agents on one machine), with all optimizations enabled

8 Related Work

At the beginning of this thesis, we have explained 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 (Tisue & Wilensky, 2004) and Repast Simphony (North et al., 2013) 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 that can take advantage of the computational power of clusters and clouds to handle large-scale simulations. We examine current simulation frameworks and explain their different computational and communication models as well as their scalability bottlenecks.

8.1 Agent-based simulation frameworks

Repast. Repast Suite is a family of agent-based modeling and simulation platforms that are widely used for research, education, and policy analysis. It is a free, open-source software package that is developed and maintained by the Repast development team at the University of Chicago's Computation Institute (Repast Developers, 2022). A widely used platform in the suite is Repast Simphony (North et al., 2013), which is a desktop-based framework for developing and executing agent-based models using Java, Python, or Groovy programming languages. Repast Simphony also includes a graphical user interface (GUI) for model development and visualization, as well as command-line tools for batch execution and analysis.

Repast Simphony does not support distributed simulations. To address this, Repast family introduces other two distributed frameworks, namely Repast HPC and Repast4Py (N. Collier & North, 2011; N. T. Collier et al., 2020), which allow for the simulation of large-scale models on high-performance computing clusters. While Repast HPC is written in C++ and mainly targets expert users, Repast4Py let users define agent classes in a more user-friendly language Python (N. T. Collier et al., 2020). The main difference between these two is the programming language; the underlying computational, communication, and programming models remain the same in both frameworks.

The computational model is based on executing *discrete events*. In the context of a simulation, discrete events represent changes to a simulation state from a variety of sources, including (1) agent actions, such as moving, interacting with other agents or the environment, or changing its state, and (2) environmental changes, such as changes in the availability of resources in an environment shared by multiple agents.

Such discrete events are stored in a global schedule queue (Repast HPC developers, 2023), and a simulation proceeds by executing such discrete events. Users can schedule the time tick at which some discrete events should occur. During the simulation, in chronological order, the scheduled event is removed from the schedule queue and executed. The global schedule queue is conceptual and can be implemented in a distributed fashion across multiple machines (N. Collier & North, 2011; N. T. Collier et al., 2020). Each machine maintains a local copy of a subset of the queue that contains only events that are scheduled to occur on that machine, which allows machines to execute events in parallel.

The communication model is built around the notion of *agent packages* in Repast HPC (Repast HPC developers, 2013), which is also referred to as *ghost agents* in Repast4Py (N. T. Collier et al., 2020). Agents are assigned to a specific process, and each process is responsible for managing its agents. Instead of agents sending messages directly to peer agents, communication occurs at the level of processes. Information about agents is moved from one process to another in the form of agent packages or ghost agents, which is a serialized representation of an agent that allows the receiving process to deserialize and reconstruct this agent. Different agent copies are synchronized manually by users.

The programming model of Repast consists of the following modeling primitives.

- "agent classes" are merely data structures that contain the state of agents and do not define any round-based or scheduled behavior. Any local computations that update the state of an agent are defined as discrete events that are scheduled to be executed for a particular discrete time stored in a global schedule queue;
- "model" defines the overall structure and behavior of the simulation. It includes information on agents, their behaviors, and interactions with each other and the environment;
- "context" is an environment shared by some agents. It includes information about agent locations and attributes, and the resources available to them. Every simulation has at least one contexts;
- "projection" describes the relationship between agents and their context. Each context has exactly one projection;
- "schedule" is used to manage the timing of events in the simulation to ensure that events are executed in the specified order and at the appropriate time.

Previous benchmark studies (Bowzer et al., 2017; Moreno et al., 2019) have shown that Repast

HPC outperforms other distributed general-purpose simulation frameworks such as FLAME (Coakley et al., 2012), due to its optimizations such as point-to-point communication and load balancing. However, compared with CloudCity, Repast HPC and Repast4Py suffer from several scalability bottlenecks in its design.

- Inefficient parallelization. Unlike CloudCity, RepastHPC does not have a notion of *agent programs* that can separate the local behavior of agents from the rest of the simulation logic. Hence, it does not take advantage of the efficient parallelization of BSP. Instead, the parallelization of Repast HPC and Repast4Py is achieved by a distributed implementation of the global schedule queue, which is less efficient than a shared-nothing system with a collection of embarrassingly-parallel agents like CloudCity. Furthermore, the programming model encompasses the concepts of shared, mutable "contexts" that allow agents to directly change shared variables. Agent events that modify these shared variables are executed sequentially in Repast HPC and Repast4Py to ensure that there is no data corruption, which makes efficient parallelization of agent events difficult.
- High synchronization cost. In the distributed implementation of the global schedule queue, each worker machine maintains a local copy of a subset of the queue. The cost for maintaining multiple copies of the schedule queue to be in sync in each iteration is much higher than the synchronization overhead of the BSP model.
- High communication cost. While messages in CloudCity contain only the required information, such as a single number, agent packages include all attributes of an agent, which is unnecessary and costly for agents that have many attributes.

FLAME. FLAME (Flexible Large-scale Agent-based Modeling Environment) (Coakley et al., 2012) is another open-source agent-based simulation framework that supports the development and execution of parallel simulations. It is designed to execute asynchronous simulations efficiently at a large scale.

In FLAME, agents are state-machine-like entities (Coakley et al., 2012; Coakley et al., 2006), where each agent has a set of variables that define its state. The transitions between states are triggered by incoming messages, which can be sent from other agents or from the environment. These messages contain information that causes the receiving agent to update its state variables and potentially generate new messages. Agents update their states asynchronously during a simulation, which means that they can respond to messages and transit to other states as soon as messages are received rather than waiting for a global synchronization event.

Specifying the states and transitions for each agent type can be a complex, error-prone, and even infeasible task, as the number of agent types and the complexity of the agents' behavior increases. To address this challenge, FLAME provides several tools and libraries that enable users to automate the generation of agent models based on high-level descriptions of the system being simulated defined in XML (Coakley et al., 2012).

Agents can interact with one another through messages, which are passed between agents to trigger actions or update state information. While agents do communicate via messaging, such messages are broadcasted to all machines (FLAME developers, 2021). This is inefficient for communication-intensive applications. Additionally, FLAME currently lacks advanced features like load-balancing. Nonetheless, FLAME remains a useful tool for researchers working on agent-based simulations, and ongoing development of the framework may address some of these limitations in the future.

D-MASON. D-MASON (Distributed Multi-Agent Simulator over Overlay Networks) is an open-source, distributed agent-based simulation framework designed to model large-scale simulations with explicit overlay networks. The communication protocol between agents depends on their relative locations in the overlay network. Each agent has a "private area", "shared area", and "halo area" (Wang et al., 2018). This allows D-MASON to exploit the spatial locality of communication patterns for better performance, and optimizes for spatial algorithms such as proximity search, which identifies neighboring agents that are located within a certain radius in a network. However, D-MASON is not a general-purpose agent-based simulation framework. In particular, agents' communication depends on their spatial attributes.

8.2 BSP-like systems

BSP-like systems are distributed systems that are based on the BSP model, offering a simple yet powerful framework for distributed computing that divides computation into a series of supersteps, where each superstep consists of three phases: computation, communication, and barrier synchronization. This model allows for efficient parallel computing by breaking the computation into smaller, manageable parts that can be executed in parallel and later combined.

Examples of popular frameworks that have adopted BSP-like models include Google's MapReduce (Dean & Ghemawat, 2004) and Pregel (Malewicz et al., 2010), making BSP model defacto standard for large-scale machine learning and graph processing. Although we have briefly described current BSP-like systems in Section 7.3.1 when cross-comparing different system designs and evaluating their relative performance, in this section, we will provide a more detailed description of BSP-like systems.

Spark. Spark is a distributed computing framework that is designed to execute iterative machine learning algorithms, such as stochastic gradient descent, over distributed data quickly and efficiently (Zaharia et al., 2012). It is built around the concept of a Resilient Distributed Dataset (RDD), which allows for data to be stored in memory across a cluster of machines. This enables Spark to perform computations much faster than traditional big data processing frameworks that rely on disk-based storage (Apache Hadoop Developers, 2006; Zaharia et al., 2012). Spark includes a wide range of built-in modules for data processing, machine learning, and graph processing, and it can be used in a variety of applications, from web analytics to scientific computing.

Spark's computational model is based on the MapReduce paradigm (Dean & Ghemawat, 2004), which can be seen as a simplified version of the BSP model, restricted to only map or reduce computations. In the BSP model, the computation is divided into a series of iterations, where each iteration consists of three phases: computation, communication, and synchronization. During the computation phase, each map or reduce worker in Spark performs local (map or reduce) computations on its own data. Then, during the communication phase, workers exchange data to update their local state. Finally, during the synchronization phase, workers wait until all other workers have completed their computations and data shuffling among workers has finished before moving on to the next iteration.

GraphX. GraphX is a graph library layered on top of Spark (Gonzalez et al., 2014) for efficient graph processing. It includes a number of built-in graph algorithms and computation primitives, such as graph aggregation, joins, and filtering, which can make it easier for users to build custom graph algorithms and applications. One of the computation primitives in GraphX is `pregel` (Gonzalez et al., 2014), which allows users to describe an iterative graph algorithm in a vertex-centric fashion (described more below when introducing Pregel).

Pregel and Giraph. Pregel is a large-scale graph processing system developed by Google (Malewicz et al., 2010). It advocates vertex-centric programming, in which the computation of a graph algorithm is performed in a series of supersteps, in the same way as BSP. During each superstep, vertices update their state based on messages received from their neighbors and then send messages to their neighbors for the next superstep. All vertices in the graph synchronize at the end of each superstep before moving on to the next one.

One of the main advantages of vertex-centric computation is its efficiency for iterative graph algorithms, such as PageRank, that require repeated updates to the vertices and edges of the graph. By optimizing computations and minimizing data movement, Pregel is able to achieve faster execution compared to more flexible models like GraphX (Gonzalez et al., 2014; Malewicz et al., 2010).

Giraph is an open-source implementation of Pregel and has been widely adopted in the industry (Ching et al., 2015; Malewicz et al., 2010). In addition to vertex-centric programming, Giraph supports a variety of built-in graph algorithms and processing primitives, making it easier for users to build custom graph algorithms and applications. Overall, Giraph is a powerful and flexible platform for processing large-scale graphs that can be used in a wide range of applications, from social network analysis to recommendation systems to bioinformatics.

Flink. Flink is a powerful distributed stream processing framework that provides a unified programming model for both batch and stream processing, enabling users to perform complex computations on data streams in a highly parallel and distributed manner (Carbone et al., 2015). This framework is designed to handle a wide range of data processing tasks, including graph processing. In particular, Flink supports BSP computation by providing a Pregel-like operator in the graph library Gelly (Developers, n.d.).

Flink's data processing model is based on a directed acyclic graph (DAG) that describes the flow of data through a computation, allowing for highly parallel and distributed processing. Flink supports two primary data processing models: the pipelined model, which enables low-latency, continuous data processing through a series of pipelined stages or operators, and the batch model, which provides high-throughput processing of batch data in discrete, batch-oriented jobs. Additionally, Flink exploits optimizations such as exploiting sparse computational dependencies via *delta* iterations (Carbone et al., 2015), which is very efficient for iterative algorithms whose results differ only partially across iterations, such as connected components (Ewen et al., 2012).

9 Conclusions and Future Work

In this thesis, 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.

A Determine K-availability from message latency

We are interested in examining whether the K-availability of an agent `A` to `B` in round t can be determined based on analyzing message latency statically. In round t, `B` may send a collection of messages to `A`, each with a possibly different latency. To ensure no message will get delayed or reordered, `A` needs to be K-available to `B` in round t, where K is the minimum message latency specified in the collection of messages sent from `B`.

However, it is not enough to just consider messages sent by `B` in round t. Recall that K-availability in round t also necessitates that `A` should be $(K-k)$-available to `B` later in round $t+k$, for $K > k$. In other words, the analysis needs to *look ahead* and consider all possible messages sent by `B` in round t as well as the next k rounds.

To illustrate, we show an example in Figure A.1. `B` may send a message to `A` in round t, with latency 10 or 5 (lines 3 – 7). To ensure that the message sent from `B` in round t arrives at `A` without being delayed, lines 3 – 7 suggest that `A` needs to synchronize with `B` in at most 5 rounds, i.e. K is at most 5. Now we need to look ahead and analyze messages sent from `B` in the interval $[t+1, t+4]$ to `A`. Lines 10 – 14 show that in the next round $t+1$, `B` sends a message to `A`, with a delay of 6 or 2. This reduces the value of K from 5 to 3 ($K-1=2$). In this example, there are no more messages sent from `B` to `A` in the interval $[t+2, t+10]$ (line 15), hence `A` should be 3-available to `B` in round t.

We point out that in the most general scenario, it is not always possible to know the message latency or the number of rounds to wait for statically, as shown in Figure A.2. The variable `latency` (line 2) is passed as an input argument rather than a constant. In such cases, K defaults to one, which causes sender and receiver agents to synchronize after just one round. Similarly, the number of rounds that an agent should wait for may be determined dynamically (line 9), then the algorithm should be conservative in the analysis and assume `wait(1)` to maximize the number of messages that needs to be analyzed when looking ahead.

```
B:
  // in round t
  if (Random.nextBoolean) {
    send(m,A,10)
  } else {
    send(m,A,5)
  }
  wait(1)
  // in round t+1, send more messages
  if (Random.nextBoolean) {
    send(m,A,6)
  } else {
    send(m,A,2)
  }
  wait(10)
```

Figure A.1: Pseudocode that illustrates a possible communication pattern from `B` to `A`.

```
B:
  latency: Int = args(0).toInt
  // in round t
  if (Random.nextBoolean) {
    send(m,A,latency)
  } else {
    send(m,A,latency+4)
  }
  wait(Random.nextInt(1, 100))
```

Figure A.2: Pseudocode of an example where messages have fixed latency, but are unknown statically.

B Dynamic Partitioning

Recall that the partition adapter in our engine supports two partition strategies, static and dynamic. The dynamic strategy is used only for $\mathscr{M}_{\mathsf{async}}$, where agents can be arbitrarily partitioned. Here we explain this strategy in detail.

For simplicity, we do not consider nested components. In particular, the partition adapter only needs to determine which agents should be moved from one component to another, for the purpose of reducing cross-component communication.[1] Due to this simplification, we assume that each worker machine contains one component. The number of components is determined by the number of worker machines in the system. In the rest of this section, we use the term "worker" interchangeably with "component".

B.1 Logical View

At a high level, re-partitioning agents is achieved via a system-level protocol between workers and the partition adapter in the driver, as illustrated in Figure B.1. The control flow is between the partition adapter and workers, shown in black arrows. The data flow is among workers, where agents are serialized and sent directly to target workers, shown in purple. Informally, a worker sends a *partition request* to the partition adapter to suggest that it wants to relocate some local agents to other workers, but such suggestions need to be proved by the partition adapter before a worker can send these agents to others. The partition adapter has the final say in how agents should be relocated, after aggregating partition requests from all workers.

We point out that the aforementioned approach depicted in Figure B.1 is a greedy heuristic that does not rely on having a holistic view of the global communication pattern that includes all agents. Instead, each worker analyzes the communication pattern of local agents and determines greedily which local agents should be moved to best reduce the worker's cross-component communication.

[1]This implicitly assumes that agents have already been separated into components. As we will explain shortly, our initial setup distributes agents evenly into different components, which is consistent with this assumption.

Our design allows a clean separation of responsibility between reducing cross-component communication and maintaining a balanced number of agents in each worker. While each worker decides how to partition local agents to minimize cross-component communication, the partition adapter is responsible for ensuring that workers have a balanced number of agents. In the following, we elaborate on how workers and the partition adapter communicate to re-partition agents dynamically.

- To change the current partition, a worker sends a partition request that contains a sequence of local agent ids that the worker wants to be moved to other workers to the partition adapter in the driver and waits for a partition confirmation from the partition adapter. A *partition request* is of the form $(W_j, (i, W'(i))_{i\in I}, c_j)$, where W_j is the current worker id, set I contains a collection of local agents in W_j that should be moved to reduce cross-component communication, and c_j denotes the number of local agents in W_i. For each i specified in I, $W'(i)$ denotes the id of the target worker that the local agent with id i should be moved to.
- Upon receiving partition requests from all workers, the partition adapter aggregates the received information and determines how agents should be moved while maintaining a balanced number of agents per worker. The partition adapter notifies each worker of the partition decision by sending a partition confirmation. The *partition confirmation* is of the form $((i, W_j, W_k)_{i\in I})$, where I contains the set of agent ids that will be re-partitioned. For each agent i that will be relocated, the partition confirmation message includes metadata such as its current location (worker W_j) and the target location (worker W_k), which are obtained from partition requests.
- After receiving a partition confirmation message from the partition adapter, workers check if any local agent is listed in the confirmation message to be relocated. If so, such agents are serialized and sent directly to their target locations. Workers also check if any new agents should be moved to them. If so, workers need to wait until the expected agents have arrived before starting the next round.

Our discussion so far has intentionally left the following questions open, viewing them as black-box instead:

- How does a worker generate a partition request? More specifically, how to decide which local agents should be moved to other components, to minimize cross-component communication?
- How does the partition adapter generate a partition confirmation, based on the received partition requests?

Subsequently, we focus on answering these two questions. We start by introducing *communication graph*, which is an abstraction used in our system to describe agents' communication

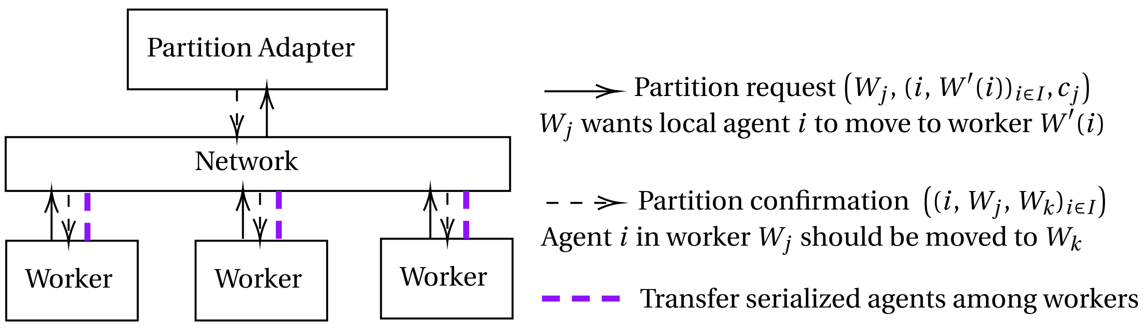

Partition Adapter
Network
Worker
Worker
Worker
Partition request $\big(W_j, (i, W'(i))_{i\in I}, c_j\big)$
W_j wants local agent i to move to worker $W'(i)$
Partition confirmation $\big((i, W_j, W_k)_{i\in I}\big)$
Agent i in worker W_j should be moved to W_k
Transfer serialized agents among workers

Figure B.1: An overview of how dynamic partition strategy can be achieved in CloudCity via communication between the partition adapter in the driver and workers. The control flow is shown as black arrows (both dotted and solid) and the data flow is shown in purple. We separate the responsibility of reducing cross-component communication and maintaining a balanced number of agents per worker: While each worker decides how to partition local agents to minimize cross-component communication, the partition adapter is responsible for ensuring that workers have approximately the same number of agents.

patterns. After that, we illustrate challenges and concerns that can arise when deciding how to generate a partition request or partition confirmation via concrete examples. We then give a thorough explanation of how our system addresses the above questions.

B.2 Communication Graph

We assume the following initial setup throughout this section. At the beginning of a simulation, agents are randomly separated into different components. The number of components is the same as the number of worker machines available in the system. Each component contains an approximately equal number of agents.

For dynamic partition, the goal is to reduce cross-component communication by dynamically re-partitioning agents while maintaining a balanced number of agents per worker. We introduce an abstraction *communication graph* to describe communication patterns quantitatively. A communication graph is weighted and undirected: A vertex in the communication graph is uniquely labeled with an agent id, and represents the agent with the same id. In particular, vertices with the same label can be duplicated in different communication graphs on separate workers, without the corresponding agents being replicated. An edge in the communication graph represents that the corresponding agents have communicated with each other. The edge weight denotes the number of messages exchanged between these two agents as observed by the worker,[2] which we make precise below when describing how a worker constructs and updates a communication graph. In our following discussion, when it is clear from the context,

[2]As we will show shortly, the quantifier "the worker has delivered" is needed because communication graphs are local and can be inconsistent among workers, where the same edge can have different weights.

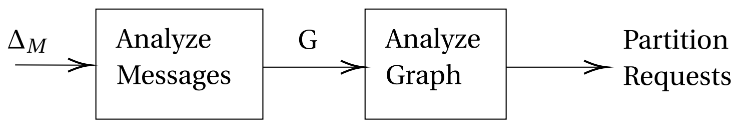


Figure B.2: High-level abstraction that summarizes how workers generate partition requests

we use the terms “vertex” and “agent” interchangeably.

Construct a communication graph from messages (construct graph). Every message can conceptually be viewed as a simple graph that contains only one undirected edge that connects the sender and receiver agents, with an edge weight of one. Therefore, building a communication graph from a sequence of messages is straightforward. Per message, we add to the communication graph vertices that correspond to the sender and receiver agents, if not already present, and increment the weight of the edge that connects the sender and receiver agents by one.

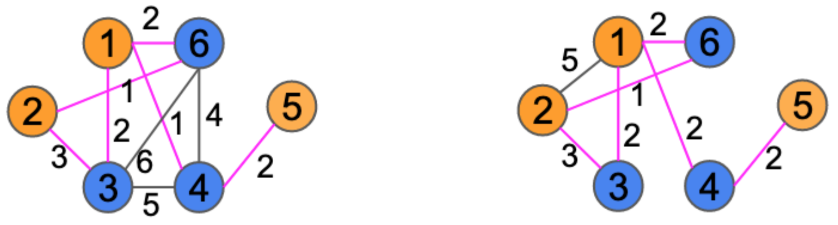
Combine different communication graphs (combine graph). Generalizing the above description of building a communication graph from messages to combining arbitrary communication graphs is straightforward because a message can be considered a special graph that contains only a single undirected edge, as we have mentioned before. Instead of incrementing the corresponding edge weight by one, to combine two edges (that connect the same sender and receiver agents) with different edge weights, we sum up these edge weights as the weight for the given edge in the combined graph.

B.3 Generate Partition Requests

From now on, we use communication graphs and the associated procedures, construct and combine graphs, as building blocks. At a high level, Figure B.2 illustrates the flow of how workers generate a partition request in each round. Below, we will explain each abstract module in Figure B.2, first describing different strategies for analyzing messages and then examining how to analyze a communication graph to generate partition requests.

B.3.1 Analyze Messages

We consider several strategies for analyzing messages, which differ in *what* messages are considered. For instance, on one end of the spectrum, *differential analysis* resembles a stateless approach that considers only messages in the current round. On the other end of the spectrum, *history-based analysis* examines all messages throughout the history of a simulation



(a) Local communication graph in Worker 1 (b) Local communication graph in Worker 2

Figure B.3: Illustration of inconsistent local communication graphs at the end of a round in (a) Worker 1 and (b) Worker 2. Orange and blue circles represent local agents in workers 1 and 2 respectively. Pink edges connect agents in different workers, and black edges connect local agents. A communication graph can contain vertices that represent agents which are not local. Black edges are unique to each worker. Pink edges are duplicated among workers, with possibly inconsistent edge weights. For instance, the edge that connects vertices 1 and 4 has weight 2 in Worker 1, but weight 1 in Worker 2.

and maintains a stateful graph across the boundary of rounds. In between, we also examine two variants of these strategies, *buffered differential analysis* and *snapshot-based analysis*. We clarify the differences between these methods later in this section.

Common to the aforementioned approaches is that in every round, a worker always analyzes all messages in the current round and updates a graph g, which we will also refer to as "local communication graph".[3] More specifically, a worker updates g as below:

- At the beginning of a round, a worker receives external messages from other workers. Per external message, the worker updates g; and
- At the end of a round, each worker collects all messages generated by local agents. For each of these messages, the worker updates g.

Note that when updating g, we do not distinguish different communication directions, such as "send" and "receive". For instance, in the case of external messages that are received at the beginning of a round, edge weights are updated due to receive events; but in the case of messages that are sent by local agents, edge weights are updated due to send events. While it is clear that an external message, either sent or received, updates the edge weight by one, we point out that a local message that is both sent and received by local agents, also increments the corresponding edge weight by exactly one, at the end of a round.

By construction, local communication graphs in different workers can be temporarily inconsistent during a simulation. If two agents in separate workers C_1 and C_2 communicate (assuming these agents have already sent messages to each other in previous rounds), then the edge

[3]Though depending on the specific technique, g can be referenced by different variable names and have different properties, including persistence across the boundary of rounds.

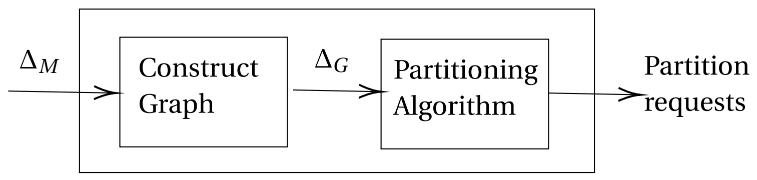


Figure B.4: Differential analysis is a message analysis strategy where the construction of a communication graph, which serves as input to the partitioning algorithm, focuses exclusively on messages within the current round.

that connects these two agents has two independent copies, in C_1 and C_2 respectively. More specifically, when a message is sent from the agent in C_1 to that in C_2, the corresponding edge in the communication graph in C_1 is updated first, at the end of the round when the send event occurs. The weight of the duplicated edge in C_2 is updated only after this message has arrived a few rounds later. In other words, the duplicated edges can be temporarily out of sync and inconsistent.

To illustrate, we show a concrete example in Figure B.3 that consists of six agents (labeled 1 to 6) partitioned into two workers. Agents 1, 2, and 5 are local agents of Worker 1, highlighted in orange, and other agents are local to Worker 2, highlighted in blue. The communication graph can include vertices that represent agents which are not local to the worker, but each vertex contains only the id of an agent. To distinguish, we use pink to denote edges that are conceptually cross-component and use black to represent edges connecting local agents. The number next to each edge represents its weight. Black edges are local to a worker. Pink edges are duplicated in different workers, with possibly inconsistent edge weights. In this example, the edge that connects vertices 1 and 4 has weight 2 in Worker 1, but weight 1 in Worker 2.

So far we have described how a local communication graph is constructed per worker in every round. We are now ready to discuss the different strategies below.

Differential analysis. The *differential analysis* considers only messages in the current round when constructing a communication graph that is later used for analysis to generate partition requests. More specifically, at the beginning of each round, an empty communication graph Δ_G is created, which fills the role of the local communication graph that we have described.

Figure B.4 shows the flow of generating partition requests under this strategy. The "Construct graph" module refers to the procedure of how to build a communication graph from a sequence of messages. We have also replaced the high-level description of "Analyze graph" in Figure B.2 with "Partitioning algorithm", which refers to a graph partitioning algorithm that takes an input graph and produces a partition request. Agents are partitioned based only on messages in the current round, as captured by Δ_G. We will describe the partitioning algorithm in more detail later.

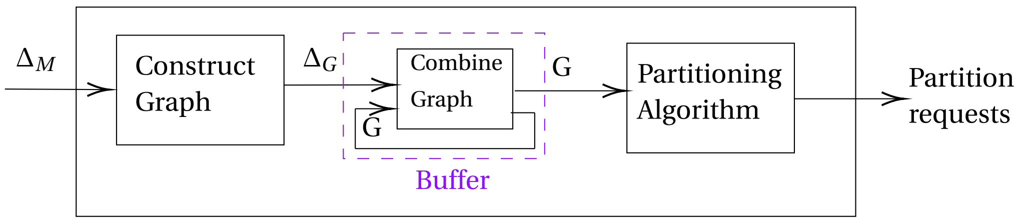


Figure B.5: Buffered differential analysis is an enhanced form of differential analysis that incorporates a buffer module to aggregate Δ_G across multiple rounds into a graph G, which is reconstructed after being used as input for the partitioning algorithm.

Buffered differential analysis. The differential approach examines only messages in a single round, but it is often desirable to consider messages over several rounds to improve the quality of partitioning by leveraging more information on communication patterns. To this end, we introduce *buffered differential analysis* that adds a buffer module to aggregate Δ_G across multiple rounds into a graph G, as highlighted in Figure B.5 in purple. Δ_G is constructed and updated in the same way as in differential analysis. The partitioning algorithm partitions G instead. After serving as input for the partitioning algorithm, G is constructed again.

The flow chart for buffered differential analysis is illustrated in Figure B.6. The decision of "Buffer Δ_G?" can be viewed as a function `bufferChanges`:

$$\texttt{bufferChanges(cond:(}w\texttt{:Worker) => Boolean):Boolean},$$

where `cond` is an anonymous function that takes the current information available in a worker and returns a Boolean variable when evaluated. If `cond` evaluates to true, then Δ_G is combined with G and the worker sends an empty partition request. Otherwise, the worker executes the partitioning algorithm with G being an input graph. Afterward, all vertices and edges in G are removed, that is, G is constructed again.

History-based analysis While differential analysis and its variant take into account only messages over recent rounds when deciding how to partition agents, an alternative strategy for message analysis is to consider the cumulative impact of all messages. We refer to this as *history-based analysis*, shown in Figure B.7. In this scenario, we introduce a graph G that persists throughout the simulation. Notably, G is not modified even after being inputted into the partitioning algorithm, unlike in buffered differential analysis.

The history-based analysis captures *long-term* information about communication patterns, whereas differential analysis and its variant contain only *short-term* information about communication patterns. To illustrate, Figure B.8 shows the communication graphs obtained under history-based analysis and buffered differential analysis respectively. We use orange and blue circles to represent local agents in workers 1 and 2 respectively. Pink edges connect

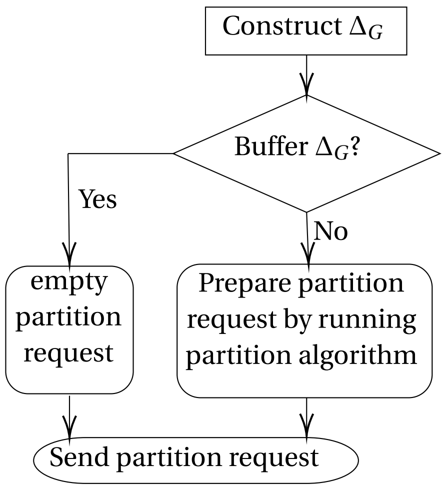


Figure B.6: A flow chart for buffered differential message analysis.

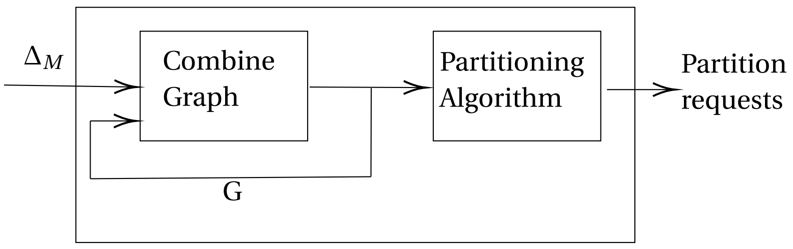
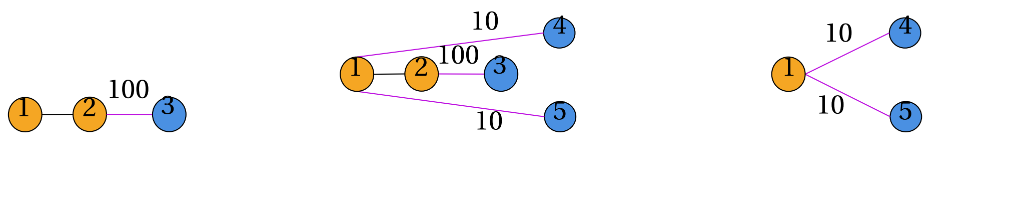


Figure B.7: History-based analysis considers the cumulative impact of all messages in history when constructing a communication graph that is inputted into the partitioning algorithm.



(a) Initial graph (b) History-based analysis (c) Buffered differential analysis

Figure B.8: Comparison between two message analysis strategies, history-based and buffered differential analysis. In round t, the communication graph is initially (a). After executing the partitioning algorithm, the communication graph in history-based analysis remains unchanged but resets to empty in buffered differential analysis. We assume that the partitioning algorithm is not executed again in the next $K+1$ rounds. In round $t+K$, the communication graph becomes (b) in history-based analysis, and (c) in buffered differential analysis. Message analysis strategies can generate distinct input graphs that lead to different partition results.

agents in different workers, and black edges connect two local agents. If unlabeled, the edge weight is assumed to be one.

In this example, we assume that local agents 1 and 2 have sent 1 message to each other, whereas agents 2 and 3 have exchanged 100 messages at the end of some round t. Furthermore, we assume that the partitioning algorithm is executed in round t, and not executed again for the next $K+1$ rounds. After running the partitioning algorithm, in history-based analysis, the communication graph is shown in Figure B.8a; in buffered differential analysis, the communication graph is reset to empty. Over the next K rounds, agent 1 exchanges 10 messages with remote agents 4 and 5 in Worker 2 respectively. The communication graph has now become Figure B.8b in history-based analysis and Figure B.8c in buffered differential analysis, which are drastically different and can lead to different partition results.

Snapshot-based analysis. A variant of history-based analysis is *snapshot-based analysis*, which separates graph update from graph partitioning by introducing a snapshot module, as shown in Figure B.9. This avoids executing the partitioning algorithm, which can be computationally intensive, in every round. We refer to this design as *snapshot-based analysis*.

Figure B.10 shows the flow chart of snapshot-based analysis. While a persistent communication graph G is updated in every round exactly like in history-based analysis, a snapshot of G is saved only periodically, which is the input for the partitioning algorithm. If there is no new snapshot, then the partition request includes an empty list.

The condition for determining whether to save a snapshot (see Figure B.10) leaves room for customization and optimization. In general, “Save a snapshot?” in Figure B.10 can be viewed as a function `saveSnapshot` that is of form

$$\texttt{saveSnapshot(cond:(}w\texttt{:Worker)=>Boolean):Boolean},$$

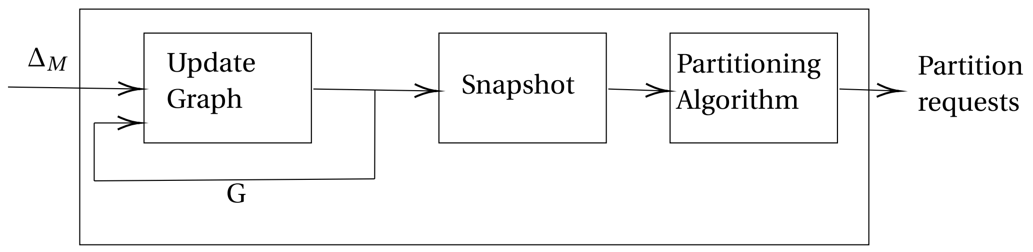
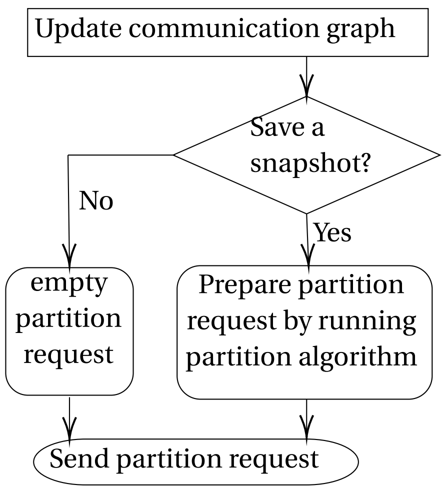


Figure B.9: Snapshot-based analysis is a variant of history-based analysis that separates graph update from graph partitioning by introducing a snapshot module.



Update communication graph
Save a snapshot?
No
Yes
empty partition request
Prepare partition request by running partition algorithm
Send partition request

Figure B.10: A flow chart for snapshot-based graph partitioning.

where `cond` is an anonymous function that takes the current information available in a worker and returns a Boolean variable when evaluated. This is similar to the function `bufferChanges` in buffered differential analysis.

B.3.2 Analyze Graphs

In the literature, graph partitioning problem has been studied under various names spanning diverse research fields, including balanced k-cut (Rangapuram et al., 2014), minimum k-cut (Chekuri et al., 2020; Guttmann-Beck & Hassin, 2000), k-partition (Cameron et al., 2008), and K-way graph partitioning (Karypis & Kumar, 1998; Kernighan & Lin, 1970). One formulation of this problem is as shown below: Given a graph $G(V, E)$ with $|V| = kn$, separate vertices in V into k components that have n vertices while minimizing the sum of edge weights for edges

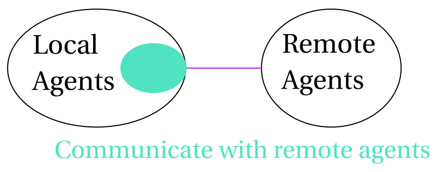


Figure B.11: Graph partitioning is difficult, but the partitioning algorithm in a worker can be simplified. In particular, vertices in a communication graph naturally form two clusters that correspond to local and remote agents. The current cross-component communication, represented by the cut between these two clusters, is shown in purple. Its cost is the sum of edge weights in the cut. We highlight the collection of local agents that communicate with remote agents using green. The goal of the partition algorithm is simplified to finding a collection of local agents such that when transforming them into remote agents, the cost of cross-component communication is reduced.

whose vertices are in different components.[4]

While graph partitioning is in general a difficult problem, in our context of partitioning a communication graph in a worker, we can simplify a partitioning algorithm due to the following factors.

- Vertices in a communication graph correspond to either local or remote agents, as shown in Figure B.11, hence naturally forming two clusters. We highlight the collection of local agents that communicate with remote agents using green.
- Cross-component communication corresponds to the cut that separates these two clusters, shown in purple in Figure B.11. The *cost* of cross-component communication is quantified by the sum of edge weights for all edges in the cut.
- The goal of the partition algorithm is simplified to finding a collection of local agents such that when moving them into other workers, hence transforming these agents into remote agents, the cost of cross-component communication is reduced.

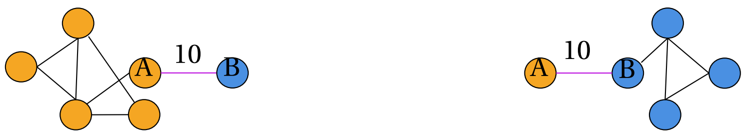
Our partitioning algorithm is based on the Linear Deterministic Greedy (LDG) algorithm (Abbas et al., 2018; Stanton & Kliot, 2012). The "linear" aspect of the algorithm refers to its time complexity, which is proportional to the size of the input data. The "deterministic" aspect of the algorithm means that it always selects the same elements in the same order, given the same input data and constraints.

We show the pseudocode of our partitioning algorithm in Figure B.12. Intuitively, we assign each local agent to the component that it communicates most frequently with. Per worker, the

[4]For $|V| \neq kn$, each component should have *roughly* equal number of vertices. For simplicity, we consider only $|V| = kn$ in our discussion.

```
1  partitionRequests = []
2  // threshold factor. Move only agents with highly skewed comm.
3  a := 4
4  for local agent v in input graph that connects with remote agents:
5    cost_i := sum of weights for edges connecting v and local agents
6    for every external component j:
7      cost_e(j) := sum of weights for edges connecting v and nodes in j
8
9    if there exists j such that cost_e(j) > (1+a)*cost_i:
10     for j that maximizes cost_e(j):
11       append (v, j) to partitionRequests
12 return partitionRequests
```

Figure B.12: Pseudocode of our partitioning algorithm



(a) Worker 1 (b) Worker 2

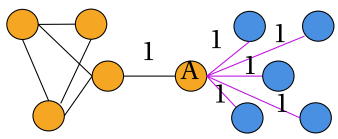
Figure B.13: Example of communication graphs at the end of a round

cost of a local agent is calculated as the difference between the number of remote messages exchanged with agents in other components and the number of local messages in the same component, multiplied by a threshold factor a (line 9). This threshold factor is introduced to restrict that only agents with highly skewed communication patterns are moved to different components. The output of this algorithm is a sequence of local agents together with their target component which should be relocated.

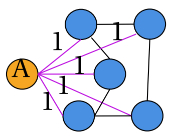
Specifically, we illustrate how our partitioning algorithm works through the following examples. For each example, agents are separated randomly into two workers, and each worker contains approximately an equal number of agents. We show the communication graph of each worker. Local agents in workers 1 and 2 are represented using orange and blue circles respectively. Pink edges connect agents in different workers, and black edges connect two local agents. If unlabeled, the edge weight is assumed to be one.

For the example shown in Figure B.13, only agent `A` is connected with remote agents (`B` in Worker 2). The internal cost of agent `A` is 1 and the external cost is 10 for Worker 2, which satisfies the condition specified on line 9 in Figure B.12. Hence, our algorithm returns [(`A`, 2)]. Similarly, in Worker 2, the partition algorithm returns [(`B`, 1)].

Clearly, directly accepting the partition requests from both workers will simply swap agents `A` and `B` in Figure B.13 into different workers, without reducing cross-component communication for any worker. To avoid this, we restrict the partition adapter to accept only one of

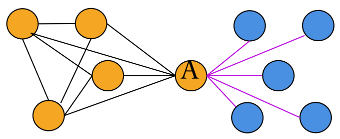


(a) Worker 1

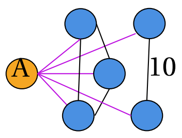


(b) Worker 2

Figure B.14: Example of communication graphs at the end of a round



(a) Worker 1



(b) Worker 2

Figure B.15: Example of communication graphs at the end of a round

such requests, if both workers want to migrate agents to each other. We will describe how the partition adapter generates partition confirmation in detail later.

Workers do not always suggest partition requests that correspond to the same cross-component edge. For instance, in Figure B.14, the partition algorithm returns [(A,2)] for the communication graph in Worker 1, but an empty list for Worker 2. Note that in Figure B.14b, the threshold factor (lines 3 and 9) suppresses vertices connected to A in Worker 2 from being partitioned. This is a tradeoff between the cost of partitioning, including serializing and sending agents, and the gain of more efficient local messaging.

We point out that our partition request messages only consider local agents that should be relocated to other workers. In Figure B.15, our algorithm returns an empty list for both workers 1 and 2. If we let workers specify which remote agents they would like to obtain, then A will be moved to Worker 2. From Figure B.15a, we see that this relocation will not reduce cross-component communication, as local communication will then become remote.

Last but not least, we consider a regular, uniform communication pattern in Figure B.16. This communication pattern is common in applications that are based on cellular automata. In this scenario, it is desirable to leave agents as they are. Thanks to the threshold factor, it is easy to verify that our algorithm returns an empty list, i.e., no vertices are suggested to be partitioned, in both workers.

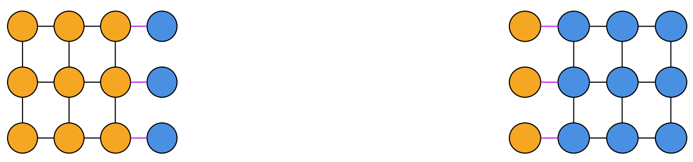

(a) Worker 1 (b) Worker 2

Figure B.16: Example of communication graphs at the end of a round

B.4 Generate Partition Confirmations

So far we have described how workers can choose different message analysis strategies to construct input graphs for our partition algorithm, which generates partition requests to reduce cross-component communication. Recall that dynamic partitioning is achieved through a system-level protocol between workers and the partition adapter. We now explain how the partition adapter resolves partition requests to determine how to relocate agents while maintaining a balanced number of agents per worker.

The partition adapter is configured with a maximum capacity threshold `max_capacity` that denotes the maximum number of agents that a worker can contain. For the partition request received from each worker W_i, the partition adapter applies the following rules:

- If W_i has reached the maximum capacity threshold, then the partition adapter rejects any partition request that proposes to move agents to W_i and accepts the partition requests from W_i to transfer agents into other workers that have not yet reached their maximum capacity.
- Otherwise, if W_i proposes to move a collection of agents $(A_i)_{i\in I}$ to W_j and W_j does not move any agent to W_i, then agents $(A_i)_{i\in I}$ are allowed to move to W_j. However, if W_j also suggests to relocate a collection of agents $(A_j)_{j\in J}$ to W_i, then either $(A_i)_{i\in I}$ or $(A_j)_{j\in J}$ are moved, but not both. This is to avoid swapping agents, as illustrated in the example in Figure B.13.
- If W_i will exceed its maximum capacity after receiving the expected agents, then arbitrarily reject some of the partition requests that are sent to W_i to ensure that the maximum capacity constraint in W_i is satisfied.

Figure B.17 shows the pseudocode for implementing the aforementioned rules in the partition adapter in the driver.

```
// Partition request
case class PartitionRequest(w: WorkerId, agentList: Iterable[AgentId,
  WorkerId], totalAgents: Int)

// Partition confirmation (agent, current worker, target worker)
case class PartitionConfirmation(agentList: Iterable[(AgentId,
  WorkerId, WorkerId)])

val max_capacity: Int // Initialized, maximum capacity per worker

val worker_capacity: Map[WorkerId, Int] = Map[WorkerId, Int]() //
  mutable map

// reject requests that want to be moved to workers that have reached
  capacity
var allowed_moves: Set[(WorkerId, WorkerId)] =
  receivedPartitionRequests.flatMap(p => {
    // populate worker capacity map
    worker_capacity(p.1) = p._3
    p.agentList.map(i => (p.w, i._2))
  }).filter(i => worker_capacity(i._2) < max_capacity)

// accept either (Ai) or (Aj) if both worker Wi and Wj want to move
  agents to each other
allowed_moves.foreach(i => {
  if (allowed_moves.contains((i._2, i._1))) {
    allowed_moves = allowed_moves.remove((i._2, i._1))
  }
})

var confirmedAgentList = receivedPartitionRequests
  .flatMap(p => p.agentList.map(i => (i._1, p.w, i._2)))
  .filter(p => allowed_moves.contains((p._2, p._3)))

// capacity constraint check, send up to max capacity
val workerSentAgents = confirmedAgentList.groupBy(_._2).mapValue(_.
  size)
val workerReceivedAgents = confirmedAgentList.groupBy(_._3).mapValue
  (_.size)

workerReceivedAgents.foreach(i => {
  val end = i._2 - workerSentAgents.getOrElse(i._1, 0)
  if (end > (max_capacity - worker_capacity)) {
    val p = confirmedAgentList.filter(j => j._3 == i).splitAt(max_
  capacity - worker_capacity - 1)
    confirmedAgentList --= p._2
  }
})

for each worker:
  send PartitionConfirmation(confirmedAgentList)
```

Figure B.17: Pseudocode of merging partition requests in the partition adapter in the driver to generate partition confirmations

C Interrupt Vector

Sometimes it is desirable to allow concurrent simulations to communicate. For instance, a user may run multiple parallel simulations to determine the best initial configuration. Similar to branch-and-bound, we can allow some simulations to halt early based on the state of other simulations, if there exists a way for different simulations to communicate while being executed.

To achieve this, we introduce *interrupt vectors* to allow drivers in different simulations to communicate, as shown in Figure C.1. Conceptually, an interrupt vector is a concurrent data structure where each driver owns a unique entry with read-write access to this data structure. A driver also has read-only access to the value of other drivers' entries.

In practice, this can be implemented by each driver maintaining an extra state that corresponds to its entry in the interrupt vector, which it can modify locally. Periodically, a copy of this entry value, together with the id of the current driver, is sent to every other driver that is running concurrently. The type of each entry in the interrupt vector is user-defined. Users also specify how a driver should update its entry in the interrupt vector via a method

```
updateInterrupt(ts : Seq[Col[Agent]], entryValue : T) : T,
```

and how to react to the value of others' entries via

```
serviceInterrupt(interruptVector : Seq[(DriverId, LocalTime, T)]) : DriverEvent,
```

where `DriverEvent` defines how the driver should react, including stopping the current simulation.

We point out that there is no synchronization among drivers. In particular, we do not require drivers to wait until receiving a copy of the interrupt vector from all concurrently running simulations before starting a new round of its own simulation. Concurrent simulations may terminate at any time and the coordination among them is done as a best-effort. Before starting a new round, each driver updates its interrupt entry by calling `updateInterrupt`,

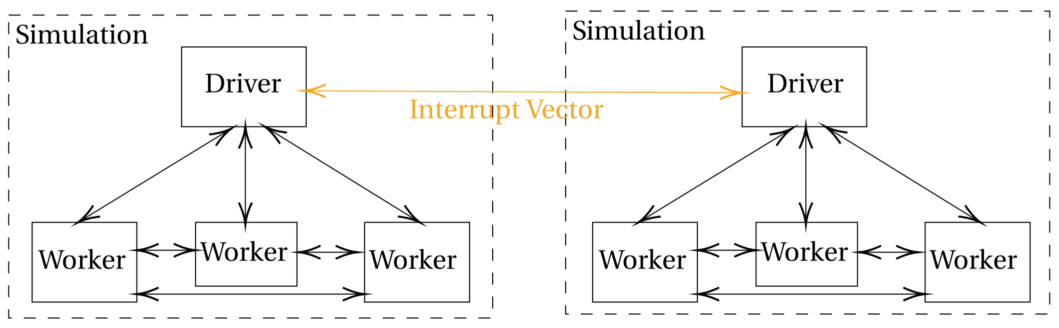


Figure C.1: Interrupt vectors allow drivers to communicate asynchronously across different simulations. Users specify the type of each entry of the interrupt vector and associated methods, such as how to update the corresponding entry of the interrupt vector and how to react to the entry values of other drivers.

sends the current entry value to other drivers, and calls `serviceInterrupt` if applicable.

Another artifact of asynchronous communication across simulations is that a driver can receive multiple copies of an interrupt vector entry from another driver. To distinguish such copies, we assume that each copy is annotated with a local clock time. Only the version with the highest local clock time is taken into consideration when applying `serviceInterrupt`.

Bibliography

Abbas, Z., Kalavri, V., Carbone, P., & Vlassov, V. (2018). Streaming graph partitioning: an experimental study. *Proc. VLDB Endow., 11*(11), 1590–1603. https://doi.org/10.14778/3236187.3236208

Adam, D. (2020). Special report: the simulations driving the world's response to covid-19. *Nature News Article*. https://www.nature.com/articles/d41586-020-01003-6

Apache Giraph Developers. (2011). Apache giraph. https://giraph.apache.org/

Apache Hadoop Developers. (2006). Apache hadoop. https://hadoop.apache.org/

Apache Spark Developers. (2018). Apache spark. https://spark.apache.org

Beran, M. (1999). Decomposable bulk synchronous parallel computers. In J. Pavelka, G. Tel, & M. Bartosek (Eds.), *SOFSEM '99* (pp. 349–359). Springer. https://doi.org/10.1007/3-540-47849-3_22

Bergstra, J. A., & Klop, J. W. (1985). Algebra of communicating processes with abstraction. *Theor. Comput. Sci., 37*, 77–121. https://doi.org/10.1016/0304-3975(85)90088-X

Bollobás, B. (2001). *Random Graphs* (2nd ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511814068

Bonorden, O., Juurlink, B. H. H., von Otte, I., & Rieping, I. (1999). The paderborn university BSP (PUB) library - design, implementation and performance. *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12-16 April 1999, San Juan, Puerto Rico, Proceedings*, 99–104. https://doi.org/10.1109/IPPS.1999.760442

Bowzer, C., Phan, B., Cohen, K., & Fukuda, M. (2017). Collision-free agent migration in spatial simulation. *Proceedings of 11th Joint Agent-Oriented Workshops in Synergy (JAWS 2017), Prague, Czech.*

Buchanan, M. (2009). Economics: meltdown modelling. *Nature, 460*(7256), 680–682.

Cameron, K., Eschen, E. M., Hoàng, C. T., & Sritharan, R. (2008). The complexity of the list partition problem for graphs. *SIAM Journal on Discrete Mathematics, 21*(4), 900–929.

Cantor, M., Shoemaker, L. G., Cabral, R. B., Flores, C. O., Varga, M., & Whitehead, H. (2015). Multilevel animal societies can emerge from cultural transmission. *Nature Communications, 6*(1), 8091.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache flink™: stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36*(4), 28–38.

Cha, H., & Lee, D. (2001). H-BSP: A hierarchical BSP computation model. *J. Supercomput.*, *18*(2), 179–200. https://doi.org/10.1023/A:1008113017444

Chattopadhyay, A. K., Kumar, T. K., & Rice, I. (2020). A social engineering model for poverty alleviation. *Nature Communications*, *11*(1), 6345.

Chekuri, C., Quanrud, K., & Xu, C. (2020). Lp relaxation and tree packing for minimum k-cut. *SIAM Journal on Discrete Mathematics*, *34*(2), 1334–1353.

Chen, J.-J., Tan, L., & Zheng, B. (2015). Agent-based model with multi-level herding for complex financial systems. *Scientific Reports*, *5*(1), 8399. https://doi.org/10.1038/srep08399

Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., & Muthukrishnan, S. (2015). One trillion edges: graph processing at facebook-scale. *Proc. VLDB Endow.*, *8*(12), 1804–1815. https://doi.org/10.14778/2824032.2824077

Coakley, S., Gheorghe, M., Holcombe, M., Chin, L. S., Worth, D., & Greenough, C. (2012). Exploitation of high performance computing in the FLAME agent-based simulation framework. In G. Min, J. Hu, L. (Liu, L. T. Yang, S. Seelam, & L. Lefèvre (Eds.), *14th IEEE international conference on high performance computing and communication & 9th IEEE international conference on embedded software and systems* (pp. 538–545). IEEE Computer Society. https://doi.org/10.1109/HPCC.2012.79

Coakley, S., Smallwood, R., & Holcombe, M. (2006). Using x-machines as a formal basis for describing agents in agent-based modelling. *Simulation Series*, *38*(2), 33.

Collier, N., & North, M. (2011). Repast hpc: a platform for large-scale agent-based modeling. *Large-Scale Computing*, 81–109.

Collier, N. T., Ozik, J., & Tatara, E. R. (2020). Experiences in developing a distributed agent-based modeling toolkit with python. *9th IEEE/ACM Workshop on Python for High-Performance and Scientific Computing, PyHPC@SC 2020, Atlanta, GA, USA, November 13, 2020*, 1–12. https://doi.org/10.1109/PyHPC51966.2020.00006

Colon, C., Hallegatte, S., & Rozenberg, J. (2021). Criticality analysis of a country's transport network via an agent-based supply chain model. *Nature Sustainability*, *4*(3), 209–215. https://doi.org/10.1038/s41893-020-00649-4

Dalziel, B. D., Novak, M., Watson, J. R., & Ellner, S. P. (2021). Collective behaviour can stabilize ecosystems. *Nature Ecology & Evolution*, *5*(10), 1435–1440. https://doi.org/10.1038/s41559-021-01517-w

Dean, J., & Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. In E. A. Brewer & P. Chen (Eds.), *6th symposium on operating system design and implementation (OSDI 2004), san francisco, california, usa, december 6-8, 2004* (pp. 137–150). USENIX Association. http://www.usenix.org/events/osdi04/tech/dean.html

Dean, J., & Ghemawat, S. (2010). Mapreduce: a flexible data processing tool. *Commun. ACM*, *53*(1), 72–77. https://doi.org/10.1145/1629175.1629198

Developers, F. G. (n.d.). Flink gelly [Accessed: 2023-03-10].

Dou, Y., Deadman, P., Berbés-Blázquez, M., Vogt, N., & Almeida, O. (2020). Pathways out of poverty through the lens of development resilience: an agent-based simulation. *Ecology and Society*, *25*(4).

Efferson, C., Vogt, S., & Fehr, E. (2020). The promise and the peril of using social influence to reverse harmful traditions. *Nature Human Behaviour*, *4*(1), 55–68.

Erdős, P., & Rényi, A. (1960). On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, *5*(1), 17–60.

Ewen, S., Tzoumas, K., Kaufmann, M., & Markl, V. (2012). Spinning fast iterative data flows. *Proc. VLDB Endow.*, *5*(11), 1268–1279. https://doi.org/10.14778/2350229.2350245

Farmer, J. D., & Foley, D. (2009). The economy needs agent-based modelling. *Nature*, *460*(7256), 685–686. https://doi.org/10.1038/460685a

Ferguson, N. M., Cummings, D. A., Cauchemez, S., Fraser, C., Riley, S., Meeyai, A., Iamsiritha-worn, S., & Burke, D. S. (2005). Strategies for containing an emerging influenza pan-demic in southeast asia. *Nature*, *437*(7056), 209–214.

FLAME developers. (2021). Flame overview [Accessed: 2021-02-09]. http://flame.ac.uk/docs/overview.html

Flanagan, C., Sabry, A., Duba, B. F., & Felleisen, M. (1993). The essence of compiling with continuations. *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 237–247. https://doi.org/10.1145/155090.155113

Fraccascia, L., Yazan, D. M., Albino, V., & Zijm, H. (2020). The role of redundancy in industrial symbiotic business development: a theoretical framework explored by agent-based simulation. *International journal of production economics*, *221*, 107471.

Gates, A., Natkovich, O., Chopra, S., Kamath, P., Narayanam, S., Olston, C., Reed, B., Srinivasan, S., & Srivastava, U. (2009). Building a highlevel dataflow system on top of MapReduce: the pig experience. *Proc. VLDB Endow.*, *2*(2), 1414–1425. https://doi.org/10.14778/1687553.1687568

Gilbert, S., & Lynch, N. A. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, *33*(2), 51–59. https://doi.org/10.1145/564585.564601

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., & Stoica, I. (2014). Graphx: graph processing in a distributed dataflow framework. In J. Flinn & H. Levy (Eds.), *11th USENIX symposium on operating systems design and implementation, OSDI '14, broomfield, co, usa, october 6-8, 2014* (pp. 599–613). USENIX Association. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

Guttmann-Beck, N., & Hassin, R. (2000). Approximation algorithms for minimum k-cut. *Algo-rithmica*, *27*, 198–207.

Hellweger, F. L., van Sebille, E., & Fredrick, N. D. (2014). Biogeographic patterns in ocean microbes emerge in a neutral agent-based model. *Science*, *345*(6202), 1346–1349. https://doi.org/10.1126/science.1254421

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, *21*(8), 666–677. https://doi.org/10.1145/359576.359585

Holland, P. W., Laskey, K. B., & Leinhardt, S. (1983). Stochastic blockmodels: first steps. *Social networks*, *5*(2), 109–137.

Iannino, V., Mocci, C., Vannocci, M., Colla, V., Caputo, A., & Ferraris, F. (2020). An event-driven agent-based simulation model for industrial processes. *Applied Sciences*, *10*(12), 4343.

Imperial College COVID-19 Response Team. (2020). *Impact of non-pharmaceutical interventions (npis) to reduce covid-19 mortality and healthcare demand* (tech. rep.). Imperial College.

Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. In P. Ferreira, T. R. Gross, & L. Veiga (Eds.), *Proceedings of the 2007 eurosys conference, lisbon, portugal, march 21-23, 2007* (pp. 59–72). ACM. https://doi.org/10.1145/1272996.1273005

Jones, D. (2015). Conflict resolution: wars without end. *Nature*, *519*(7542), 148–150.

Karypis, G., & Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, *20*(1), 359–392.

Kernighan, B. W., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, *49*(2), 291–307. https://doi.org/10.1002/j.1538-7305.1970.tb01770.x

Kersting, M., Bossert, A., Sörensen, L., Wacker, B., & Schlüter, J. C. (2021). Predicting effectiveness of countermeasures during the covid-19 outbreak in south africa using agent-based simulation. *Humanities and Social Sciences Communications*, *8*(1), 174. https://doi.org/10.1057/s41599-021-00830-w

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, *21*(7), 558–565. https://doi.org/10.1145/359545.359563

Lim, M., Metzler, R., & Bar-Yam, Y. (2007). Global pattern formation and ethnic/cultural violence. *Science*, *317*(5844), 1540–1544. https://doi.org/10.1126/science.1142734

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In A. K. Elmagarmid & D. Agrawal (Eds.), *Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2010, indianapolis, indiana, usa, june 6-10, 2010* (pp. 135–146). ACM. https://doi.org/10.1145/1807167.1807184

Mark, W. R., Glanville, R. S., Akeley, K., & Kilgard, M. J. (2003). Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, *22*(3), 896–907. https://doi.org/10.1145/882262.882362

Marlin, C. D. (1980). *Coroutines: A programming methodology, a language design and an implementation* (Vol. 95). Springer. https://doi.org/10.1007/3-540-10256-6

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

Milner, R. (1980). *A calculus of communicating systems* (Vol. 92). Springer. https://doi.org/10.1007/3-540-10235-3

Milner, R. (1999). *Communicating and mobile systems - the pi-calculus*. Cambridge University Press.

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

Moreno, A., Rodríguez, J. J., Beltrán, D., Sikora, A., Jorba, J., & César, E. (2019). Designing a benchmark for the performance evaluation of agent-based simulation applications on HPC. *J. Supercomput.*, *75*(3), 1524–1550. https://doi.org/10.1007/s11227-018-2688-8

Moura, A. L. D., & Ierusalimschy, R. (2009). Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, *31*(2). https://doi.org/10.1145/1462166.1462167

Mukherjee, U. K., Bose, S., Ivanov, A., Souyris, S., Seshadri, S., Sridhar, P., Watkins, R., & Xu, Y. (2021). Evaluation of reopening strategies for educational institutions during covid-19 through agent based simulation. *Scientific Reports*, *11*(1), 6264. https://doi.org/10.1038/s41598-021-84192-y

Netlogo dictionary [Accessed: 2021-02-09]. (n.d.).

Niamir, L., Kiesewetter, G., Wagner, F., Schöpp, W., Filatova, T., Voinov, A., & Bressers, H. (2020). Assessing the macroeconomic impacts of individual behavioral changes on carbon emissions. *Climatic Change*, *158*(2), 141–160.

North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M. J., & Sydelko, P. (2013). Complex adaptive systems modeling with repast simphony. *Complex Adapt. Syst. Model.*, *1*, 3. https://doi.org/10.1186/2194-3206-1-3

Orach, K., Duit, A., & Schlüter, M. (2020). Sustainable natural resource governance under interest group competition in policy-making. *Nature Human Behaviour*, *4*(9), 898–909. https://doi.org/10.1038/s41562-020-0885-y

Pal, C., Leon, F., Paprzycki, M., & Ganzha, M. (2020). A review of platforms for the development of agent systems. *CoRR*, *abs/2007.08961*. https://arxiv.org/abs/2007.08961

Palmer, R., Brian Arthur, W., Holland, J. H., LeBaron, B., & Tayler, P. (1994). Artificial economic life: a simple model of a stockmarket. *Physica D: Nonlinear Phenomena*, *75*(1), 264–274. https://doi.org/https://doi.org/10.1016/0167-2789(94)90287-9

Parreaux, L., & Shaikhha, A. (2020). Multi-stage programming in the large with staged classes. *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 35–49. https://doi.org/10.1145/3425898.3426961

Parreaux, L., Shaikhha, A., & Koch, C. E. (2017). Squid: type-safe, hygienic, and reusable quasiquotes. *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, 56–66.

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

Rai, V., & Henry, A. D. (2016). Agent-based modelling of consumer energy choices. *Nature Climate Change*, *6*(6), 556–562.

Rangapuram, S. S., Mudrakarta, P. K., & Hein, M. (2014). Tight continuous relaxation of the balanced k-cut problem. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 27: annual conference on neural information processing systems 2014, december 8-13 2014,* *montreal, quebec, canada* (pp. 3131–3139). https://proceedings.neurips.cc/paper/2014/hash/f60bb6bb4c96d4df93c51bd69dcc15a0-Abstract.html

Repast Developers. (2022). Repastsuite. https://repast.github.io/

Repast HPC developers. (2013). Repast hpc reference manual. https://repast.github.io/hpc_tutorial/RepastHPC_Demo_01_Step_07.html

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

Schelling, T. C. (1969). Models of segregation. *The American economic review, 59*(2), 488–493.

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

Silverman, E., Gostoli, U., Picascia, S., Almagor, J., McCann, M., Shawm, R., & Angione, C. (2021). Situating agent-based modelling in population health research. *Emerging Themes in Epidemiology, 18*.

The sir model for spread of disease - the differential equation model [Accessed: 2023-05-24]. (2004).

Stanton, I., & Kliot, G. (2012). Streaming graph partitioning for large distributed graphs. In Q. Yang, D. Agarwal, & J. Pei (Eds.), *The 18th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '12, beijing, china, august 12-16, 2012* (pp. 1222–1230). ACM. https://doi.org/10.1145/2339530.2339722

Tang, L., Wu, J., Yu, L., & Bao, Q. (2015). Carbon emissions trading scheme exploration in China: a multi-agent-based model. *Energy Policy, 81*, 152–169. https://doi.org/https://doi.org/10.1016/j.enpol.2015.02.032

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., & Murthy, R. (2009). Hive - A warehousing solution over a map-reduce framework. *Proc. VLDB Endow., 2*(2), 1626–1629. https://doi.org/10.14778/1687553.1687609

Tian, Z., Lindner, P., Nissl, M., Koch, C., & Tannen, V. (2023). Generalizing bulk-synchronous parallel processing model: from data to threads and agent-based simulations. *Proc. ACM. Management of Data (PACMMOD)*, 439–480. https://doi.org/10.1145/3589296

Tisue, S., & Wilensky, U. (2004). Netlogo: a simple environment for modeling complexity. *International conference on complex systems, 21*, 16–21.

Torre, P. d. l., & Kruskal, C. P. (1996). Submachine locality in the bulk synchronous setting (extended abstract). *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II, 1124*, 352–358. https://doi.org/10.1007/BFb0024723

Trauer, J. M., Lydeamore, M. J., Dalton, G. W., Pilcher, D., Meehan, M. T., McBryde, E. S., Cheng, A. C., Sutton, B., & Ragonnet, R. (2021). Understanding how victoria, australia gained control of its second covid-19 wave. *Nature Communications, 12*(1), 6266. https://doi.org/10.1038/s41467-021-26558-4

Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM, 33*(8), 103–111. https://doi.org/10.1145/79173.79181

Wadler, P. (1988). Deforestation: transforming programs to eliminate trees. In H. Ganzinger (Ed.), *ESOP '88, 2nd european symposium on programming* (pp. 344–358). Springer. https://doi.org/10.1007/3-540-19027-9_23

Waldrop, M. M. (2018). What if a nuke goes off in washington, d.c.? *Science News Article*. https://www.science.org/content/article/what-if-nuke-goes-washington-dc-simulations-artificial-societies-help-planners-cope

Wang, H., Wei, E., Simon, R., Luke, S., Crooks, A., Freelan, D., & Spagnuolo, C. (2018). Scalability in the MASON multi-agent simulation system. *22nd IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2018, Madrid, Spain, October 15-17, 2018*, 135–144. https://doi.org/10.1109/DISTRA.2018.8601006

Wu, J., Wang, X., & Pan, B. (2019). Agent-based simulations of china inbound tourism network. *Scientific Reports*, *9*(1), 12325.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In S. D. Gribble & D. Katabi (Eds.), *Proceedings of the 9th USENIX symposium on networked systems design and implementation, NSDI 2012, san jose, ca, usa, april 25-27, 2012* (pp. 15–28). USENIX Association. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

Zoellner, C., Jennings, R., Wiedmann, M., & Ivanek, R. (2019). Enable: an agent-based model to understand listeria dynamics in food processing facilities. *Scientific Reports*, *9*(1), 495.

Zilu Tian

zilu.tian@epfl.ch

Education:

EPFL, Switzerland, Sept. 2017 Ph.D. candidate in Computer Science (EDIC)
Sept. 2017 - Sept. 2023

Worcester Polytechnic Institute (WPI), U.S. B.S. in ECE and Math
Aug. 2013 - May 2017

Research experience:

Distributed agent-based simulation engine, EPFL, Sept. 2020-present
- Generalized Bulk-synchronous parallel (BSP) computational model for agent-based simulations
- Designed a domain-specific language (DSL) for the parallel programming of agents
- Implemented a distributed engine for large-scale simulations based on Akka
- Evaluated various optimizations, such as transforming messaging to direct memory accesses, de-parallelizing threads, and deforestation, to improve distributed executions
- Publication *Generalizing Bulk-Synchronous Parallel Processing for Data Science: From Data To Threads and Agent-Based Simulations*, SIGMOD'23

Hardware accelerator for data transformation, EPFL, May 2018 – Nov. 2019
- Participated in designing a hardware accelerator for removing data ser/des overhead in RPCs
- Publication *Optimus Prime: Accelerating Data Transformation in Servers,* ASPLOS'20
- Patent, 6.1969-US, 07.27.2019

Evaluation of Arm servers using a cloud-native benchmark, EPFL, Sept. 2017 - May 2018
- Performed detailed microarchitecture profiling for ARM using CloudSuite (Cavium ThunderX)
- Identified software scalability bottleneck in server workloads across different layers
- Working knowledge of fault-tolerant distributed framework (Spark, Hadoop, HDFS, MapReduce), database (Cassandra, MySQL), data caching (Memcached), web serving (NGINX), open-source web search (Solr/Nutch)
- Working knowledge of virtualization technology such as Docker and Kubernetes
- Detailed knowledge of Perf, VTune, and FlameGraph
- Presented CloudSuite in a conference tutorial in ISPASS'18, Belfast, UK

Course projects:

Machine learning, EPFL
- Identified road segments in satellite images using convolutional neural network (CNN) and others

Formal verification, EPFL
- Implemented a satisfiability solver (SAT) prototype with optimizations such as implication graphs

TA experience / supervised projects:

Database Systems, Introduction to Computer Architecture, Analysis I-IV
NetLogo compiler, Agriculture simulation, Traffic simulation, Online data visualization

Languages: Scala, Java, Python, Bash, C/C++