SAPIENZA
UNIVERSITÀ DI ROMA

An Agent-Based Simulation API for Speculative PDES Runtime Environments

Facoltà di Ingegneria

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Andrea Piccione
ID number 1422045

Thesis Advisor

Prof. Alessandro Pellegrini

Co-Advisor

Prof. Roberto Beraldi

Academic Year 2018/2019

Thesis defended on 22 October 2019
in front of a Board of Examiners composed by:

Prof. Giuseppe De Giacomo (chairman)

Prof. Aris Anagnostopoulos

Dr. Silvia Bonomi

Dr. Daniele Cono D'Elia

Prof. Domenico Lembo

Prof. Luca Iocchi

Dr. Alessandro Pellegrini

Prof. Aurelio Uncini

An Agent-Based Simulation API for Speculative PDES Runtime Environments

Master's thesis. Sapienza – University of Rome

© 2019 Andrea Piccione. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: October 9, 2019

Author's email: piccione.1422045@studenti.uniroma1.it

Dedicated to

Tania

Abstract

Agent-Based Modeling and Simulation is an effective paradigm to model systems which exhibit complex interactions. The goal is studying them in the hope of devising their emergent behavior, if it exists. Applications of this methodology range from modeling agent decisions in the stock market, supply chains, and consumer markets, to predicting the spread of epidemics, the threat of bio-warfare, and the factors responsible for the fall of ancient civilizations.

While Agent-Based Modeling and Simulation has been effectively used in many disciplines, most successful models are still run only sequentially, causing a potential waste of the computing resources offered by modern multi-core architectures. The high reliance of the model developers community on simple and easy-to-use languages such as NetLogo places a limit on the possibility to benefit from more effective runtime paradigms, such as Parallel Discrete Event Simulation (PDES). This is a significant problem since the required size of Agent-Based Models simulations is increasing everyday: traditional implementations are not up to the challenge.

The aim of this thesis is to somewhat bridge the gap between efficient simulation runtime paradigms, in particular Speculative PDES, and Agent-Based Modeling and Simulation. For this purpose we propose a semantically-rich API which allows to implement Agent-Based Models in a simple and effective way.

We also describe the critical points which should be taken into account when implementing this API in a speculative Parallel Discrete Event Simulation environment, in order to scale up simulations on distributed massively-parallel clusters. We include in this thesis a description of the implementation we developed, with a focus on the various optimizations we devised.

Our experimental assessment, carried on our reference implementation, shows that our API allows to implement complex interactions between agents and the surrounding environment with a reduced complexity, while delivering

a non-negligible performance increase. This is a first important step in finally making powerful simulation tools accessible to the practitioners, independently of their computer science knowledge.

Acknowledgments

Throughout the writing of this thesis I have received a great deal of support and assistance. I would first like to thank my thesis advisor, Dr. Alessandro Pellegrini, whose expertise was invaluable in formulating the thesis topic and methodology. I also want to thank my co-advisor, Prof. Roberto Beraldi for the useful feedback and for his patience with me.

I would like to acknowledge my colleagues in Lockless for the wonderful working experience they provide me with everyday. You supported me greatly and were always willing to help me. I want to thank you for all of the opportunities I was given.

In addition, I would like to thank my parents for their wise counsel and sympathetic ear. You are always there for me.

Finally, I want to thank my friends, who were of fundamental support in providing a happy distraction to rest my mind outside of my work.

Contents

Chapter 1

Introduction

A *simulation* is the imitation of the behavior of a real-world or hypothetical system, computed over time. The process of developing and studying a simulation can be roughly described in two steps:

1. Devise and develop a simulation *model*, the set of assumptions that captures the relations between the entities in the system of interest. This step requires expertise in the system domain in order to formulate meaningful and reasonable assertions.

2. Evaluate the model on an appropriate time span; for very simple models this evaluation may be conducted computing a closed form mathematical solution. More often, computer-based simulation are used to run the model. This way, data over time is produced as if we were observing the real system.

Usually in the process, those two steps are reiterated multiple times, until the model has been satisfyingly tuned for the required accuracy and efficiency.

Historically, the first example of "manual" simulation is the Buffon *needle experiment* in 1777 [21]. The experiment consists in throwing needles onto a plane with equally spaced parallel lines in order to estimate the value of π. We have to wait for the spread of general purpose computer in the 50s to see

the first actual computer simulations. This shows that this field is relatively recent, research in the area is interesting and active.

Some advantages and applications of simulation practices are:

- Test new policies, operating procedures, decision rules, information flows, but also physical layouts, hardware designs, transportation systems and so on without impacting the real system we are supposedly trying to improve.

- Study aforementioned systems without real world time constraints: simulations can be sped up or slowed down allowing respectively a reduction of the human effort and time needed to examine a process or the possibility of taking precise measures which could be otherwise impossible to take on real world systems.

- Answer *what if* questions regardless of the real feasibility of tested changes. This is helpful, for example, in devising new strategies for disaster recovery.

- Improve the understanding of complex systems, for example by studying system's sensibility to initial conditions or by observing the relations between variables throughout simulation runs.

An interesting and problematic aspect of simulation is that it's an intrinsically multidisciplinary field. Knowledge of the targeted system domain is necessary in order to formulate meaningful and appropriate models, but also computer science know-how is required since simulations need to run on computers. Domain experts are often unable to code their models from scratch, either for a lack of IT knowledge or for convenience reasons. Therefore they usually rely on well known modeling paradigms for which simulation frameworks are available . Such software suites often provide the developers with a easy to use custom computer language, or a graphical development tool. The model can

then be run leveraging the runtime environment included in the framework. This thesis focuses at first on one popular simulation paradigm in particular: *Agent-Based Modeling*.

Agent-Based Modeling (ABM) is a powerful simulation modeling paradigm in which the system is represented by a collection of autonomous decision-making entities (the agents) which are set out in an environment [3, 32]. Each agent individually assesses the surrounding environment, also taking into account the presence of other agents, and makes decisions on the basis of a certain set of rules which implement their behavior. During its lifetime, an agent can decide to change its behavior, also depending on the environment state and interactions with other agents. The actions that agents take might also have effects on other agents and/or on the surrounding environment—for example, an agent can *produce*, *consume*, or *exchange* items.

ABM is considered incredibly powerful for multiple applications and real-world business problems for a number of reasons. First of all, the model developer can concentrate on the design of the behavior of different agents independently of *where* the agents will act. This significantly simplifies the development of complex simulation models, allowing to reach results which could be difficult to reach relying on more traditional mathematical methods [15, 11]. Second, the interaction of multiple agents in a system can exhibit complex behavioral patterns [44], able also to show (or even anticipate) what is commonly referred to as *emergent behavior*. Even a simple agent-based model, executed on a large enough scale, can exhibit complex behavior patterns [44] and provide valuable information about the dynamics of the real-world system that it emulates. Several approaches have also coupled sophisticated models with neural networks [20], evolutionary algorithms [31], or other learning techniques in order to provide the agents with behavioral adaptation, making ABM even more powerful and realistic.

In general, agent-based models require three common aspects to be dealt

with to be effectively used to find a solution for real-world problems:

- *Agent specification*: each agent is described in terms of its *simulation state*, namely the set of explanatory variables which keep track of its evolution and interaction with the environment/other agents; this state might drive the behavior of the agent during the simulation. Each agent must be identified uniquely in the whole system.

- *Interaction definition*: agents interact with the environment and with other agents. The decisions which agents make might depend also on the state of portions of the environment which are nearby the current position of the agent, or on the presence/absence of other specific agents in the (surrounding) environment.

- *Topology specification*: ABM is extremely versatile, therefore requiring to model complex topologies. While many approaches rely on grid-based environments, more complex (graph-based) topologies can enable more interesting interactions to be described or to emerge. Modern ABM requires to overcome the traditional definition of a physical environment, because it might be defined by several layers of (also virtual) information, and might therefore be also modified by the agents themselves.

The size of problems tackled through ABM is growing in size at an unprecedented pace. The intrinsic dynamic nature of real-world phenomena commonly modeled through the ABM formalism can easily lead to a significantly increased complexity due to the following reasons (also in combination):

1. A very large number of agents are involved, or the environment is extremely large. Here, unfeasibility can materialize either in too long execution times, or in non-sufficient memory capacity on single nodes (the so-called *memory wall* [33]). This is a fundamental aspect especially when large ABMs are necessary to disclose an emergent behavior which

a small-scale variant of the model is not able to show, e.g. when the emergent property is linked to the amount of interactions between the agents [34, 46].

2. Increased amount of data which enable the implementation of micro-level simulation models [36], based on the inclusion of an always larger set of explanatory variables in the state of single agents.

3. Enriched decision-making policies in the agents, such as rational behavior or cognitive and psychological processes. These aspects (mostly related to micro-simulations) can require higher computing demands.

4. Interest in systems which exhibit a non-linear, dynamic behavior, with high uncertainty and notable degree of stochasticity. In this context, a single run of the simulation model is not sufficient: to obtain statistically-significant results, a calibration based on multiple repeated simulations is the only solution to deal with parameter sweep. While a single simulation could be feasible, a large number of simulations might be not.

While the ABM formalism allows to easily deal with this increased complexity, traditional frameworks used to run ABM-based simulations might experience performance penalties which make finding solutions to these *what-if* analysis problems unfeasible, preventing researchers to find the needed insights [2]. As an example, in contexts such as demography [36], ABM applications are developed mostly relying on sequentially-executed languages such as NetLogo [49]. This choice is driven by the fact that developing sequential models is easy, especially for experts coming from domains not related to computer science. This is a problem so hot that researchers are investigating multiple approaches to speedup the execution of agent-based models on extremely parallel architectures [56].

Before introducing our solution to this problem, we shortly describe another simulation paradigm: Discrete-Event Simulation (DES). This paradigm aims

at describing the behavior of a system as a sequence of discrete events in time. In DES models, the values of variables between two events are not relevant for system dynamics or they result trivial to be computed in case of necessity. In simulation jargon, the acting entities are named *Logical Processes* (LPs) and they interact by processing and sending events to each other. The standard implementation of a DES runtime keeps the events ordered in a queue by their timestamp and processes those in a serial fashion.

DES has seen wide applications in health care systems, for example to model the processes in a hospital with the aim of optimizing them. It also has been used in economy to model potential investments or to model the stock market in an attempt to predict them. Probably though, computer network simulation has been DES most successful application, and justifiably so: routers and computers fit well under the LP concept and the data packets they exchange are perfectly modeled by discrete events.

From an architectural point of view, Discrete-Event Simulation (DES) is the paradigm which best fits the runtime requirements of ABM, since the impulsive nature of the events proper of DES perfectly embraces the way agent-based models should be executed. To meet the current scale of these models, the literature on Parallel DES (PDES) [17] has identified in the speculative Time Warp synchronization protocol [25] a viable solution to cope with large-scale parallelism.

In speculative PDES, LPs concurrently execute their events without waiting to synchronize their virtual time with each other. This maximizes computing power utilization but corrective actions are necessary in order to preserve correctness. Time Warp enforces consistency by means of the *rollback operation*. Any time that it is determined that the execution has reached an inconsistent portion of the simulation, e.g. due to the reception of an out-of-order event, a previous consistent snapshot is identified and restored, allowing the trajectory to deviate and resolve the causality violation. Logical time synchronization is

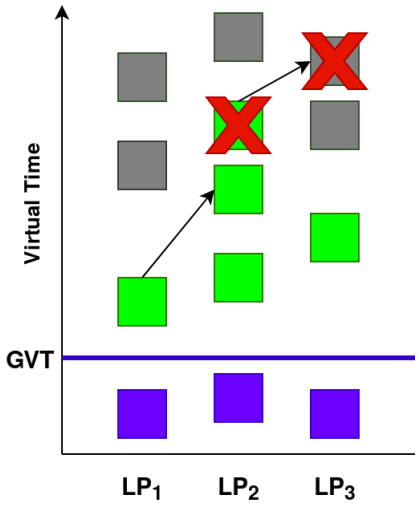

Figure 1.1. LP_1 sends a straggler message to LP_2, the events crossed in red need to be retracted

enforced only periodically in order to detect termination, diminish frequency of rollbacks and to free up the memory used to store the snapshots. The rollback operation has been traditionally implemented either by means of state saving and restore (since the seminal paper in [25]), or by means of reverse computation [5]. In the former case, the simulation model developer is often exposed to the state saving operation, either in terms of the invocation of explicit services exposed by the runtime environment to take a checkpoint, or by the need to annotate or notify the runtime environment of the organization in memory of the simulation state. In the latter case, the modeler is often asked to implement by hand *reverse events*, which at least doubles the time and cost of model development, although some proposals in the literature have tried to automatize this process (see, e.g., [54, 29, 8]).

Anyhow, when dealing with speculative PDES, the modeler might be exposed to many details which should not be mastered by domain experts. In particular when dealing with the large number of domains which benefit from ABM, it is simply not possible to demand from the developers to explicitly tackle all the aforementioned technical aspects. We propose a solution to this problem:

this thesis describes an API explicitly targeting the development of agent-based models, which has been devised keeping in mind the three aforementioned needs of ABMs (namely, agent specification, interaction definition, and topology specification). This API is meant to be semantically-rich, thus allowing domain experts to benefit from it in the process of defining an agent-based model. At the same time, it has been designed also keeping in mind the peculiarities of speculative PDES run on distributed clusters of massively-parallel machines, hiding away from the modeler its complexity. Indications to implement this API in speculative runtime environments are described in this work in an attempt at bridging the gap between the users of ABM runtime environments, and the architectural capabilities of modern computing infrastructures. We evaluate both the effects on programmability, and the performance of an implementation of the API, to show the viability and effectiveness of our proposal.

The remainder of this thesis is organized as follows. In Section 1.1 we discuss related work. Section 2 presents the API which we have devised to simplify the implementation of ABMs on top of last-generation PDES runtime environments. In Section 3 we discuss the salient details of our reference implementation. An assessment of our proposal, both in terms of effects on programmability and performance of an implementation of the API is discussed in Section 4.

1.1 Related Work

ABM and PDES are two topics which have been extensively studied in the literature, and there have already been attempts to bridge the gap between the two worlds. In particular, we can find a large number of frameworks to support agent-based simulation in the literature. For a thorough comparison of different ABM frameworks, we refer the reader to the comprehensive work in [1]. Here, we discuss only proposals which have the closest relation to our

work.

The MASON framework [30] pays special attention to the performance of simulation execution, addressing computing-intensive models (i.e., scenarios with many agents), along with portability and reproducibility of the results across different hardware architectures. A parallel/distributed version (D-MASON) has been presented in [12], which relies on time-stepped synchronization and on the master/slave paradigm. We similarly address the performance of agent-based simulation execution, yet we do this for the case of speculative asynchronous (non-time-stepped) PDES. In particular, we benefit from the performance improvement which can stem from the Time Warp synchronization protocol, while allowing a simple implementation of agent-based models via an expressive API.

Pandora [55] is a C++-based simulation framework enabling executions in parallel/distributed environments. It features several AI algorithms for supporting agents' decision making and provides python bindings (which is a benefit for inexperienced programmers). At the same time, Pandora does not hide its internal complexity by design, allowing (and sometimes requiring) the model developer to extend its fundamental classes, just to perform simple actions. Conversely, in our proposal we offer the simulation model developer an API that is specifically tailored for implementing agent-based models, and which hides away all the idiosyncrasies of synchronization in a distributed simulation. This allows for a simplified implementation of simulation models, giving transparently access to highly optimized synchronization facilities to support efficient computations on clusters of modern parallel machines.

AnyLogic [27] is a commercial multi-method general-purpose simulation modeling and execution framework, offering at the same time the possibility to support discrete-event, system dynamics, and agent-based simulation. The simulation model developer can rely on graphical modeling languages to implement the simulation models, along with Java code. Differently from

this framework, we target the simplicity of agent-based model development by means of an API which can be effectively exploited by multiple domain experts, while delivering high-performance simulations thanks to an effective synchronization protocol in its underlying runtime support. Moreover, the possibility to transparently deal with an arbitrary topology which can change at runtime is an aspect which makes our proposal different from the work in [27] and mostly unique, to the best of our knowledge.

FLAME [23] is a simulation framework targeting large, complex models with large agent populations to be run on HPC platforms using MPI and OpenMP. The counterpart FLAME GPU [45] targets 3D simulations of complex systems with a multi-massive number of agents on GPU devices. We keep the ability to deal with large amount of agents, yet we rely on traditional CPU-based execution of the simulation model.

RepastHPC [10] and Swarm [24] are two ABM-oriented runtime environments which have been successfully used to deliver high performance of agent-based models. These runtime environments support different programming languages, and allow agents to interact through the exchange of discrete events. Differently from our proposal, anyhow, they require high programming skills to be effectively used. Therefore, they are commonly regarded as complex-to-use frameworks [1].

RAMSES [7] is a runtime environment explicitly targeting ABM, with a focus on transparency. An ABM-oriented API has already been proposed in [7], with a goal similar in spirit with that of our proposal. The main differences between the two works are that: i) in [7], the API is based on the implementation of complex functions which are passed via pointers to the API, making it difficult to create bindings in different languages; ii) RAMSES has been implemented only with reverse computation supports, while the API which we propose can be implemented in both reverse computation-based and state saving-based runtime environments; iii) if an agent has to take decisions

based on the state of the surrounding environment, this has to be implemented via explicit message passing; iv) dynamic topologies are not supported.

Chapter 2

The ABM API

As mentioned earlier, we propose an API for ABM which is semantically-rich for the developers and which, at the same time, targets a runtime implementation to transparently bridge the gap towards efficient speculative PDES runs. To this end, before discussing the API, we introduce the reference system model of runtime environments which could support our API.

2.1 Reference System Model

The reference system architecture which we target in our proposal is the Symmetric Multi-Threaded optimistic simulation kernel, which was introduced in [53] for distributed massively multi-core architectures. This system architecture is depicted in Figure 2.1, and is basically a multi-layer organization. On top of everything we find Logical Processes (LPs), which are handled by a simulation kernel instance. LPs are identified with a unique ID in the range $[0, numLPs - 1]$, where $numLPs$ is the total number of active LPs. In case LPs should be identified with other (unique) attributes (e.g., literal names), a mapping function to the range of integers could always be applied.

The simulation kernel instance is devised according to the multi-threaded programming paradigm, and at simulation startup it takes control up to all

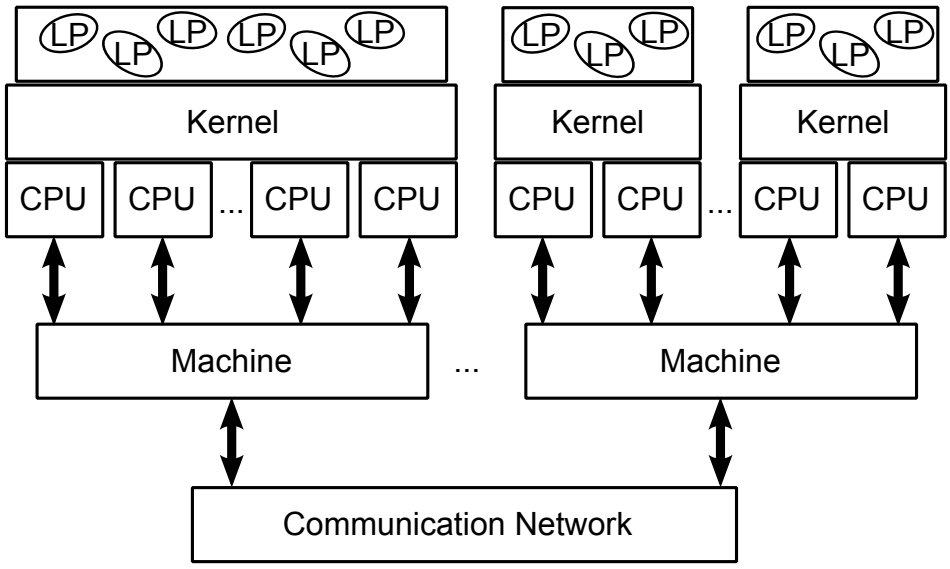

Figure 2.1. System Architecture: Symmetric Multi-Threaded Simulation Kernel.

the available CPU cores available on the host machine, by spawning a worker thread for each used core.

Multiple machines could be involved in the simulation, thus supporting distributed and parallel discrete event simulations. The same per-machine organization, with one multi-threaded simulation kernel instance taking control up multiple CPU cores, is actuated on every node, which are then connected via a message passing-based communication network (e.g., by relying on an MPI library [37]). Every worker thread selects the LP to schedule according to a *smallest-timestamp first* (STF) policy, across a subset of (temporarily) bound LPs

Each LP has a couple of queues, similarly to the original Time Warp proposal in [25]. Namely, every LP has an *input queue*, to store in timestamp order all the events which are received. Executed events are not discarded, but a reference to the last-correctly executed event (the *bound*) is kept. Therefore, the input queue is (logically) both a future-event list, and a past-event list. No specific implementation of the queue is required by the symmetric multi-threaded organization, therefore the input queue can be organized as a linked

list, a skip list [42], a calendar queue [4], a splay tree [48], a ladder queue [13], or any other data structure tailored for high-performance management of events. A similar organization regards the *output queue*, which is again a per-LP data structure, which is used to record the sending of a message. This information is used, in case of a rollback operation, to send *antimessages*, if needed.

Similarly to the original Time Warp proposal in [25], the symmetric multi-threaded organization requires that rollbacks are executed lazily. In particular, whenever a LP receives a straggler message, the information that it has to rollback is stored somewhere (e.g., in its execution context), so that the actual rollback operation takes place only when the STF scheduler detects that the LP to be executed is the one that has to rollback. This can significantly reduce the rollback frequency.

There is no specific requirement on the Global Virtual Time (GVT) algorithm to be selected for the commitment of events in the symmetric multi-threaded organization. Specifically, either blocking [18], non-blocking [39], or asynchronous [51] algorithms can be used. The only requirement is that the GVT reduction is executed at least periodically, and that when a new GVT value is adopted, additional housekeeping operations can be hooked.

2.2 Modeling Agents and their Interactions

We describe here all the functions exposed by the API, also describing what are the aspects that an implementation within any (speculative) runtime environment should take into account to ensure performance and consistency of the simulation. The API is presented as signatures of a possible C implementation, but by no means it is limited to this programming language. As a preliminary note, this is a stateful API: therefore, the runtime environment must maintain a set of data structures to keep track of the evolution of the simulation. For the sake of performance, we suggest that these data structures are setup lazily,

i.e. when a first call to the API requiring them is issued.

Taking inspiration from [7, 40], and taking also into account the way agent-based models are commonly implemented, in our programming model we map environmental regions to LPs, while agents are mapped to data structures (or objects, in runtime environments supporting object-oriented programming). Agents are therefore handled as program variable, a simple concept which is mastered also by non-experience programmers from a multitude of domains. Additionally, this is a fundamental choice, because it can significantly simplify the implementation of multiple agents' interactions in a neighboring region. Moreover, although in ABM we talk of the environment, by no means ABM is limited to physical environments. Therefore, it is possible that the "regions" of the environment represent logical entities, and this aspect must be taken into account by the runtime environment—this is exactly why we do not force developers to be stuck, e.g., to grid-based environments, but we support also generic (graph-based) environments—for the sake of simplicity of style, we will talk of *regions* in the remainder of this thesis, also when referring to a node in an abstract graph topology.

The first aspect to take into account is how to identify an agent. To this end, we introduce the `agent_t` type, which can be regarded as a *unique identifier* (UID) describing a single agent system-wide. The runtime environment should be able, also in a distributed setting, to generate `agent_t`s uniquely. This is mostly a trivial aspect in our system model. Indeed, each simulation kernel instance is already identified uniquely in the system (e.g., thanks to its MPI rank). Therefore, it is sufficient to rely on a per-instance monotonic counter and generate UIDs by computing the *Cantor pairing function*:

$$\frac{(K+c)(K+c+1)}{2}+c \tag{2.1}$$

where K is the id of the simulation kernel instance (e.g., it's MPI rank) and c is the current value of the per-instance monotonic counter. This approach is quite effective—the pairing function is fast, as it relies only on integer arithmetic—

and can work mostly out of the box in a multithreaded environment thanks to hardware synchronization facilities such as the *fetch and add* read-modify-write instruction[1], which allows all threads to consistently read and increment c also in concurrent execution[2].

An efficient management of the `agent_t` type is fundamental, because agents are dynamic entities. As we will show, agents can have a lifetime shorter than that of the whole simulation, as we explicitly allow new agents to be spawned or removed from the simulation at any time. Therefore, any time that the model developer wants to interact with agents, they must be provided with API functions which allow to manage dynamically handled `agent_t`s. Therefore, we suggest to organize `agent_t`s hosted within a certain portion of the environment (an LP) into a hash map, so as to reduce the latency of retrieving the data structures used to represent an agent. It is important to note that, as mentioned, the state of an agent must keep all the explanatory variables that allow the agent's logic to take decisions or change its own behavior. To this end, the simulation model developer can rely on any arbitrarily-complex data structure, and associate that at runtime with an `agent_t`. Since in an agent-based model there could be different families of agents—in the limit case, each agent might be described by a different simulation state—the simulation model developer must be allowed to easily define a per-agent data structure.

2.2.1 Basic Agent Management

After these considerations, we can now start to describe the part of the API which allows to manage the agents in the model. The following API functions are supported.

[1] This is a hardware facility which is available on all commodity CPU implementations.

[2] This approach can of course return two equal values when the counter overflows, but relying on 64-bit integers will significantly reduce this probability.

`SpawnAgent(unsigned user_data_size)` This is the API call which allows, at runtime, to create a new agent in the system. It returns an `agent_t` uniquely identifying the new agent system-wide. `user_data_size` tells the size (in bytes) of the space which should be allocated and associated with the returned `agent_t`, so as to allow the model developer to store and carry around data. Given the speculative nature of the simulation support which we envisage, the creation of an agent should be rollbackable. In particular, this creation should be rolled back if the event during whose execution the invocation to `SpawnAgent()` took place is undone. Therefore, the runtime environment must associate the (dynamically-allocated) buffers to keep the agent representation with the event which caused its allocation and the LP at which this event has been executed.

The literature already offers multiple solutions which can be used to solve this problem (see, e.g., [50, 41]). In particular, if the runtime environment allows the simulation model to rely on dynamic memory (e.g., by supporting calls to `malloc()`/`free()` which are transparently rolled back), then it is sufficient to redirect the allocation of the agent data structure to per-LP memory. This means that, for a limited time span, the agent belongs to the simulation state of the LP modeling the region win which the agent is residing. This is clearly desirable, since all the interactions between agents which happen within one region of the environment (mapped to an LP) will belong (for a period of simulated time) to the same simulation state. In this way, all the agents will observe a consistent simulation snapshot while simulation events access/modify their data structures.

In this way, if the event which created a new agent is rolled back, the policies taken to restore a previous consistent simulation state will release the memory associated with that agent too. Given that agents are identified by a UID, if the UID-generating module applies a rule similar to Equation (2.1), there is no need to rollback the state of that generator—simply, the UID

associated with the rolled back agent will not be associated with any agent in the corrected simulation trajectory. An exception to this consideration happens if the simulation runtime uses silent events execution to restore a state from a previous checkpoint. In this case the rollback of the UID generator is necessary in order to keep consistency with possible events sent by agents spawned during the silently executed events.

`KillAgent(agent_t agent)` Similarly to the creation of an agent, the simulation model might determine at a certain point in simulation that an agent no longer has to be part of the agent. The `KillAgent()` API serves this purpose. This API function accepts an `agent_t` generated by a previous call to `SpawnAgent()`. The semantic associated with this API is such that, from that point in simulation on, any attempt to interact by API calls with the agent identified by the UID stored in `agent` should fail.

Regarding the release of the memory buffers used to keep track of that agent, we note that the event issuing a call to `KillAgent()` can be also rolled back. Therefore, the invocation to this API function is subject to rollbacks as well. With respect to the buffers used to keep the agent data structure, the same considerations made for the `SpawnAgent()` API function apply. On the other hand, the situation is different with respect to the UID kept in `agent_t`. Indeed, any subsequent interaction with that agent (even in the same event) should fail. Yet, if the call to `KillAgent()` is rolled back, that UID must become valid again.

As mentioned, we suggest to organize all UIDs in a fast hash table. The hash table can be augmented with a per-agent flag, telling whether some UID has been retracted from the simulation or not. A call to `KillAgent()` will therefore simply flag an agent as retracted. The UIDs which are retracted during the execution of the event can be stored in an ad-hoc buffer (e.g., a resizable array) kept in the data structure describing the event. Upon a rollback

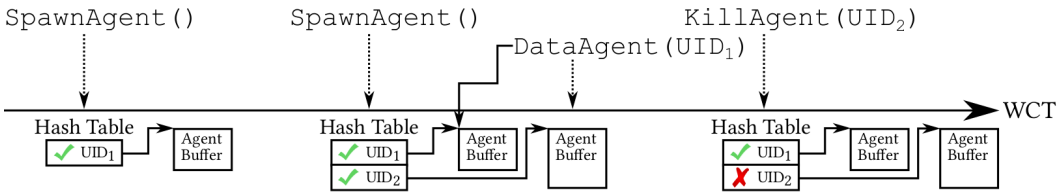

Figure 2.2. Agents management timeline.

operation, the message buffer of the undone event is inspected, and all agents which were retracted can be "reintroduced" in the system by simply clearing their retraction flag. An agent is definitely (and consistently) removed from the simulation upon the GVT computation. In particular, when a committed event is removed from the LP queue (as in traditional Time Warp simulations), its retracted agents' buffer can be inspected so as to remove from the hash map the associated UIDs, and free memory buffers.

`DataAgent(agent_t agent)` Since the state of an agent, composed of its explanatory variables, can be freely organized by the simulation model developer, we have devised a simple API to manage agents' states. This API function, given the UID of one agent, returns a pointer to a memory buffer which can be used to store the agent's state. The size of this memory buffer should be at least of `user_data_size` bytes, i.e. the amount specified when creating an agent via `SpawnAgent()`. This memory buffer must be handled by the runtime environment from its allocation, via a call to `SpawnAgent()`, to its dismissal, after the computation of the GVT.

An overall timeline of the lifetime of an agent in one region (LP) of the simulation is depicted in Figure 2.2.

2.2.2 Supporting Agent Interactions and Decisions

We now move to describing the part of the API which allows multiple agents to interact when they are in the same portion of the environment (i.e., the same LP), or in proximity.

`CountAgents(void)` As mentioned, ABM can be particularly useful to study emergent behaviors. One fundamental aspect, when implementing agent-based models, is to know the number of agent which are close to each other. Since in our system model each LP manages a portion of the environment, we can easily retrieve the number of agents which are currently in the LP running a simulation event by calling this API function.

The implementation of this API function should be quite simple, provided that there is a fast per-LP hash table, telling what are the UIDs of the agents currently registered at one LP.

`IterAgents(agent_t *agent_p)` In order to let the simulation model developer to easily scan through all the agents which are registered at a certain LP, this API function allows to implement an iterator. In particular, by repeatedly calling this function, the model will find stored in `agent_p` the id of the "next" agent registered in the region. Once the UID of the "next" agent is retrieved, the simulation model can interact with it in the desired way, by relying on any other API function. There is no strict requirement on the order according to which agents' UIDs are returned to the simulation model developer, therefore the most efficient implementation can be picked for the runtime environment—this strongly depends on the way agents are registered within one region: if a per-LP hash table is used, then the "most efficient order" could be that of scanning through the table.

`TrackNeighbourInfo(void *neighbour_data)` More complex decisions can be taken by the agents if they knew about the state of the environment in proximity, not only of the portion of the environment in which they currently reside. While an agent-based model run in a sequential or time-stepped environment can immediately access this information, speculative PDES poses an additional challenge. Indeed, in our system model, different regions are mapped to different LPs, which might have reached different simulation time

instants due to the speculative nature of the simulation. Supporting this decision-making capability can be burdensome for the system. Consider, as an example, an agent registered in a region mapped to LP_x which has to decide whether to reach either LP_y or LP_z depending on the "crowdiness" of the destination regions. To make an informed decision, the agent has first to collect the number of agents currently residing at LP_y and LP_z. To implement this logic in traditional PDES, the model developer has to split this logic into multiple events to acquire the agent count in both LPs and then determine what is the correct destination. Given the speculative nature of the simulation, LP_y and/or LP_z might be forced to rollback. Additionally, since these "support events" could likely be *simultaneous events*, this might place an additional burden to the runtime environment, due to the need for some sort of *tie-breaking* function.

`TrackNeighbourInfo()` is an API function which tries to solve these problems. The idea behind this API function is to implement a sort of publish/subscribe protocol between connected LPs. The model developer, at simulation startup, can define a portion of its simulation state which should be broadcast to all neighbor LPs, according to the current topology of the simulation—we will describe how the topology is supposed to evolve later in this paper. In this way, the runtime environment is requested to make a local copy of the data of interest across neighboring LPs. This copy, which can be broadcast via traditional message passing, is superseded every time that a new version of the data is installed. Therefore, any time that an event needs to access data from a neighbor, a local copy can be inspected.

Two points deserve a discussion about an efficient implementation of this API function. First, not all events might update the portion of the state which is watched by the `TrackNeighbourInfo()` API. Therefore, to reduce the number of messages which are silently exchanged to support this API, the runtime environment must detect whether the watched memory area has been

updated by an event or not. To this end, depending on the actual organization of the runtime environment, we envisage two possible baseline implementations. One implementation, which is the one which we have used in the experimental assessment presented in this paper, relies on a fast hash function to compute a hash of this memory buffer[3] in order to detect whether the last-executed event modified the buffer. An additional option, which is more costly, is to rely on memory protection mechanisms provided by, e.g., `mprotect()`.

The second point is associated with consistency of the values observed by the neighbors. Since updates must be transparently sent to neighbor LPs upon a state update, if the original event which caused the state transition is rolled back, also the updates towards the neighbor LPs should be annihilated. This is especially important because if an event executed at a neighbor LP read the retired value, also its execution must be rolled back, according to the global synchronization mechanism proper of Time Warp simulations. To this end, we suggest to rely on *control messages*, namely messages which are exchanged at the level of the runtime environment, destined to a certain LP, that are included in the event queue but are never delivered to simulation model handlers. These control messages allow to record that at a certain point in the simulation trajectory some relevant event happened, which must be subject to rollback operations, or which can be subject to the delivery of antimessages. The logic associated with these events is exactly that of updating the local copy of remote portions of the simulation state, once the timestamp associated with them is reached in the simulation. Annihilating such control messages via anti events cancels copy updates which have not yet been processed, while relying on the rollback operation allows to restore previous consistent snapshots in case that remote state updates are retracted.

[3]We suggest using xxHash [9], an extremely-fast general-purpose hashing function working closely to the speed of RAM, and faster than `memcpy()`. Implementations for multiple programming languages are available.

Of course, the buffers to store these copies must be rollbackable, so the same considerations for the buffers to keep agents explanatory variables apply.

`GetNeighbourInfo(direction_t direction, unsigned int *region_id, void **data_p)` This API function is a simple entry point to the transparent local copies made by the runtime environment thanks to the `TrackNeighbourInfo()` API function described above. The goal of this API is to retrieve the data of a neighboring LP, which is made accessible by the runtime environment via the `data_p` pointer. To select what is the region which the used is interested in, `the direction_t` data type can be used. This allows to "navigate" the selected current environment topology, as we shall discuss later. In `region_id`, the runtime environment stores the id of the LP currently handling the portion of the environment which we target with `direction`.

2.2.3 Moving Agents Around

We now discuss the portion of the API which supports agents' mobility. To fully understand the functions which we have devised in our API, we shall make a discussion on why usually time-stepped simulation is considered easy-to-use in agent based simulations. In many classical example agent-based models from the literature (e.g., flocks [44], or game of life [19]), updates in the agents and/or the environment happen "globally". This means that it is not possible to take a definitive decision observing a subregion of the model. This approach, although simple, significantly clashes with the speculative PDES data partition-oriented approach, which is one of the reasons why the Time Warp synchronization protocol is so effective. To bridge the gap between these two approaches, we model agents' mobility in a "best-effort" manner. This means that, once the simulation model logic decides that an agent should move from one region to another, this is the "best educated guess" that the model can make at that point of the simulation. This guess can be proven

wrong later on by the model itself, and therefore the runtime engine should be informed of a modified decision—this also means that the mobility decision can be retracted completely.

To support this programming model without placing a high burden on the developer, we adopt a twofold strategy. On the one hand, we introduce the concept of *visits*: an agent has a set of regions to be visited, which can be modified at any time. These create a *visit list* which allows to describe the "ordering of places" in the topology which the agent will reach during the lifetime of the simulation—we emphasize again that the topology can be "virtual", meaning that the visit list can be also used to construct at runtime a set of pending logical tasks for the agent or a combination of physical and logical actions. Every time that the agent takes a decision, it can modify this list arbitrarily. Two different agents have always a different visit list, but of course two agents can reach the same regions in the same order, depending on the logic that the simulation model is implementing. This is a versatile way to describe the way an agent explores and interacts in the environment, which can be easily adapted to many real-world problems.

`EnqueueVisit(agent_t agent, unsigned region, unsigned event_type)` This API function allows to store at the end of the current visit list a new region to be reached. Once this visit is fired—we will discuss later how and when this happens—the destination LP identified by the id `region` is hit by the `event_type` event.

`CountVisits(const agent_t agent)` This API function allows to return the number of regions which are stored in the visit list which can be reached by the agent in its future moves. Past visits, namely regions which have already been visited, are not accounted for in this number. The `CountPastVisits(const agent_t agent)` API function can be used to this end.

`GetVisit(const agent_t agent, unsigned *region_p, unsigned *event_type_p, unsigned i)` During the lifetime of the model, it might be of interest to inspect the future visit list. To this end, with respect to the `i`-th future visit, a call to this API will store into `region_p` the id of the LP to which the visit is associated, and into `event_type_p` the event which will be scheduled when this visit is fired. If a user is interested in getting information about past visits, the `GetPastVisit(const agent_t agent, unsigned *region_p, unsigned *event_type_p, simtime_t *time_p, unsigned i)` API function can be used.

With respect to the runtime environment supporting this API, we emphasize that the `GetPastVisit()` API function can produce a non-negligible memory footprint to be supported. Indeed, if the agents have a high mobility, especially in very long simulations, this past visit list might become very long. Since the semantic of this API is to allow to retrieve any past visit, also those associated with the committed portion of the simulation trajectory, we are not allowed to prune the past visit list even when a new GVT value is computed. Due to this consideration, we leave to the implementation the choice to always maintain the list of past visits, or start keeping them once the first invocation to `GetPastVisit()` is issued—in this latter case, of course, the model might not be able to observe the whole past visit list.

The last API functions to manage visit lists should allow to modify the current list. `AddVisit(agent_t agent, unsigned region, unsigned event_type, unsigned i)` allows to insert a new visit just before the `i`-th element in the future visit list, while `SetVisit(const agent_t agent, unsigned region, unsigned event_type, unsigned i)` allows to modify an already inserted visit (e.g., to modify the event associated with the firing of that visit). If a visit is to be removed from the future list, `RemoveVisit(agent_t agent, unsigned i)` can be used.

In Figure 2.3 we provide a graphical representation of the visit list and

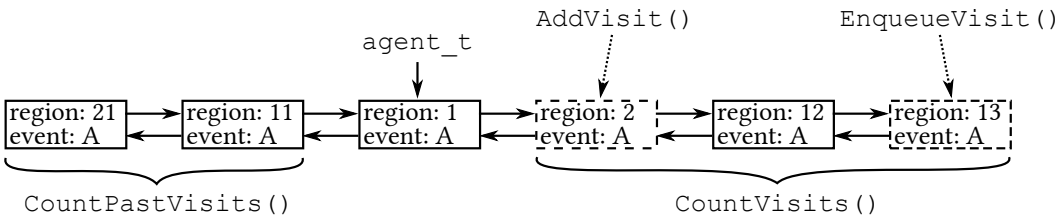

Figure 2.3. Visit management with respect to current region.

the management operations supported by our API. From an implementation point of view, the visit list can be realized either by relying on resizable arrays, or by using actual linked lists. While the latter implementation is easier, we encourage to pick the former, because it can provide a significant performance improvement (thanks to memory locality). Moreover, such a compact data structure might be easily migrated around, e.g. across different NUMA nodes, thus being highly manageable by environments which enforce memory-oriented optimizations at runtime.

`ScheduleNewLeaveEvent(simtime_t time, unsigned int event_type, agent_t agent)` A visit list by itself does not identify at what simulation time the agent will reach the next region. To this end, the `ScheduleNewLeaveEvent()` API function allows to tell the runtime environment what is the current guess for the simulation time at which the "next hop" in the visit list will be taken. The simulation model is allowed to call this API function multiple times, therefore allowing to override the previous guess, provided that this call is issued by an event executed by the LP which is currently hosting the agent and before the visit event is fired—conversely, it will fail. We note that if this call fails, it should not produce any runtime error, as the model developer might want to rely on this simple semantic (namely, "try" to make an agent leave) to keep the implementation simpler. If an agent is removed from the simulation by means of a call to
KillAgent(), the firing of the leave event will never take place.

The only constraint in this API function which should be enforced is that

`time` is in the future of the current simulation time at which the call takes place—this is in compliance with the traditional Time Warp synchronization protocol. To support this API function, we suggest the runtime environment to rely once again on *control messages*. These should be placed in the event queue of the LP which has called the `ScheduleNewLeaveEvent()` so that when its clock reaches the point in the simulation execution identified by `time`, then an event is sent to the LP associated with the next region to be visited. The payload of this event is the data structure (also with the user-defined payload) associated with the agent whose UID is `agent`.

We note that this leave event can be rolled back as any other event. For the sake of efficiency, we suggest not to remove the data structure associated with `agent` from the sender LP, but rather to use the retraction flag which we have discussed before. These buffers will be released upong the computation of the GVT. While this might overall consume more memory, especially if an agent moves a lot, it can significantly reduce the overhead to support rollback operations. We envisage memory recollections policies, similar in spirit to the *cancelback protocol* [26], such that if free available memory falls below a certain threshold, buffers keeping older incarnations of agents which have left an LP can be released.

As a last note, we have to reconcile the just discussed optimization with the possibility that an agent visits the same region multiple times. In this case, it could be that in the speculative portion of the simulation trajectory, the same UID entry in the hash table should be associated with multiple incarnations of the same agent. In this case, it is sufficient to extend the hash table with a multiversion list of agents. In this list, the agents could be organized in descending timestamp order, thus allowing simulation events to always find at the head of the multiversion list the newest incarnation of the agent. If a rollback operation occurs, then the no-longer consistent nodes—these are the agents incarnations which have been rolled back—can be unchained and

freed. In this way, again, the head of the multiversion list will point to the last consistent snapshot of the agent representation.

2.3 Describing the Topology

As mentioned, the topology definition in an ABM-oriented API must guarantee two fundamental aspects, along with extreme simplicity: i) high expressiveness, due to the fact that the ABM paradigm allows to represent very different aspects of the world; ii) the possibility to arbitrarily change the topology at runtime. As for the second point, this is fundamental due to the fact that physical environments can be modified by the agents, or can be subject to changes related to the evolution of the agents (e.g., in disaster or rescue scenarios). Supporting a changing topology in a speculative runtime environment is a challenging aspect.

An initial topology must be specified at simulation startup. This aspect is trivial and well studied in the literature, we therefore comply with traditional file-based startup configurations—in our implementation, we rely on a JSON file[4]. The following is the set of topologies which we consider fundamental for ABM and which should be supported by the runtime environment, by means of the proposed API:

- *Square*: Each LP models a square cell. The overall environment can be either a square or a rectangle (this poses a limitation on the number of LPs that can be used in a simulation run). Four or eight neighbors can be reached from each cell, except from the boundaries.

- *Hexagon*: Each LP models a hexagonal cell. This topology is similar in spirit with the Square topology, but 6 neighboring cells can be reached

[4]For the lack of space, we cannot provide the full specification of the JSON file which we have envisaged. Anyhow, several complete configuration files are available at the online repository.

from every cell not lying on the border.

- *Ring*: LPs are organized in a mono-directional ring. Agents can move from one LP to only one adjacent LP, in a single direction.
- *Bid-Ring*: A bidirectional ring is similar in spirit to a Ring topology, except for the fact that agents can move in two directions.
- *Torus*: A torus is similar in spirit to a Bid-Ring topology, except for the fact that this is a 3D topology in which from each LP there are 4 possible neighbors to reach.
- *Star*: A single LP is connected to all LPs. If an agent wants to move to any other LP, it has to pass through the central LP.
- *Fully Connected Mesh*: Every LP is connected to any other LP. This topology can create the highest burden on the runtime environment, due to the high level of communication that can exist across the network.
- *Graph*: This is a generic weighted graph. Each LP is connected to a subset of all the LPs, and each connection (an edge) is associated either with a weight or with a probability. The agents can stochastically move around, taking into accounts the weights or the probabilities on each edge. In this case, for each node, the startup configuration file should provide a list of reachable nodes, with the associated weight/probability.

The latter topology is exactly the one which allows to model changing environments at runtime. In particular, the graph topology can be implemented as a weighted fully-connected mesh, in which forbidden connections have an infinite weight or a zero probability. In this way, by relying on the topology API, it is possible to change at runtime these weights/probabilities, in order to modify the connections across the environmental regions. We note that changes in the topology must be again rollbackable. Therefore, the runtime

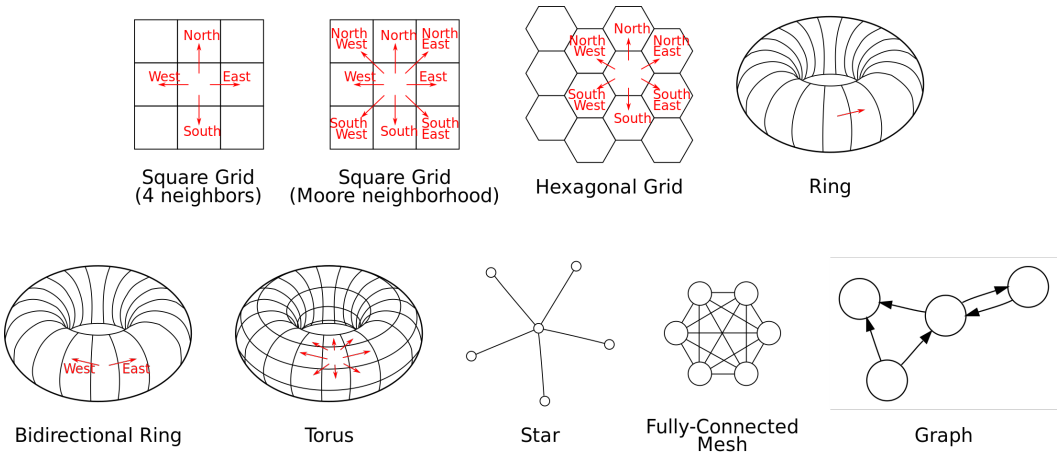

Figure 2.4. Graphical representation of the topologies

environment must associate the event which altered the topology with the new incarnation of the topology, and in case of a rollback, the change must be undone. We note that this is not a trivial aspect, given that other events scheduled at other LPs might depend on a no-longer consistent topology. These events must be also rolled back, thus possibly generating cascading rollbacks. Moreover, changes in the topology might target LPs hosted at a remote kernel instance in a distributed environment, which must be notified of these changes via message passing. Again, we suggest to split the variables describing the overall topology at each LP in a scattered way, relying on control messages to mark updates in it. Provided that these control messages are incorporated in the event queue of each LP, traditional rollback mechanisms allow to keep the overall topology consistent.

To let the model developer alter the environment and fruitfully navigate it, we have devised the following API functions. `NeighboursCount(unsigned int region_id)` and `RegionsCount(void)` allow to retrieve from the current organization of the topology, respectively, the number of regions which can be reached from the one which is executing the event performing the call, and the total number of active cells (the total number of LPs). This is useful, e.g., to determine the reachability degree of a region, or to implement actions

to observe and sense the surrounding environment. `NeighboursCount()` is especially useful on boundary regions, which might not be able to let agents navigate in all directions provided for the current topology (e.g., in a square topology, central regions can move in four directions, but edge cells have fewer legal moves).

In case of a graph-based topology, the weights or the probabilities can be modified by relying on the `SetValueTopology(unsigned int from, unsigned int to, double value)` API function. This function updates the `value` associated with the edge between the `from` and `to` LPs. We emphasize again that both these LPs can be different from the one running the event causing the invocation of this function. Therefore, control messages should be used to enforce a causally consistent update at the `from` LP. The counterpart API function is `GetValueTopology(unsigned int from, unsigned int to)`, which allows to query the current value for the topology.

To allow the model developer to simply code the movements of the agent, it is fundamental to offer API functions which allow to cleverly query the current topology. In case of traditional topologies (i.e., different from graph-based), the concept of *direction* is intrinsic: in a grid-like environment the agent has a limited set of possible moves. `GetReceiver(unsigned int from, direction_t direction)` returns the id of the neighbor which would be reached by moving `direction`-bound. As an example, in the case of a square topology, valid values for `direction` are *north*, *south*, *east*, *west*.

At the same time, since ABM is commonly a stochastic simulation, we also envisage the `FindReceiver(void)` API function which, depending on the current topology, picks a random neighbor. In grid-based topologies, the probability distribution should be considered uniform. On the other hand, for graph-based topologies, the distribution should account for the probabilities or weights associated with the edges.

The last two API functions associated with the topology deal with a

more long-termed planning. `FindReceiverToward(unsigned int to)` returns the id of the next LP to visit in order to reach the LP identified by `to`. This API should consider the minimum-cost path, either considering the number of moves, or the weights in case of a graph-based topology. In our reference implementation we have relied on Dijkstra's pathfinding algorithm [14]. To extract the complete tour with all visits, the modeler can rely on `ComputeMinTour(unsigned int source, unsigned int dest, unsigned int result[RegionsCount()])`, which returns an array keeping a list of LPs to visit. In this latter case, the model developer must take into account the fact that this list might change in the future of the simulation, because other concurrent LPs might alter the topology, as we have discussed. Nevertheless, this API function (used in conjunction with the concept of visit lists which we have described before) allows to prepare a tentative schedule for the actions of an agent in the immediate future—it can be modified at any time with all the API functions which we have described so far.

The `FindReceiver()` function interprets topology weights as proportional values to traversal probabilities while `ComputeMinTour()` and `FindReceiverToward()` consider weights as traversal costs. We expect that developers aren't interested in mixing within the same model those two incompatible interpretations. We therefore distinguish between *cost-based topologies* and *probability-based topologies*.

Chapter 3

Implementation

We have implemented the API and its associated runtime support in the ROOT-Sim runtime environment [38]. This choice has been made for various reasons:

- ROOT-Sim adheres with the requirements in Section 2.1
- ROOT-Sim already deals with the synchronization problems arising in massively parallel architectures; this aided us in substantially reducing the work needed with respect to implementing a solution from scratch.
- ROOT-Sim is a stable and well written software since it has been around for three decades. Comprehensive documentation of the internals is also available.
- ROOT-Sim is open source. This is important so that this thesis can be a useful work for others. The source code of ROOT-Sim and this API implementation is freely available on *Github* at `https://github.com/HPDCS/ROOT-Sim`.

We briefly describe ROOT-Sim main characteristics before illustrating our API implementation details.

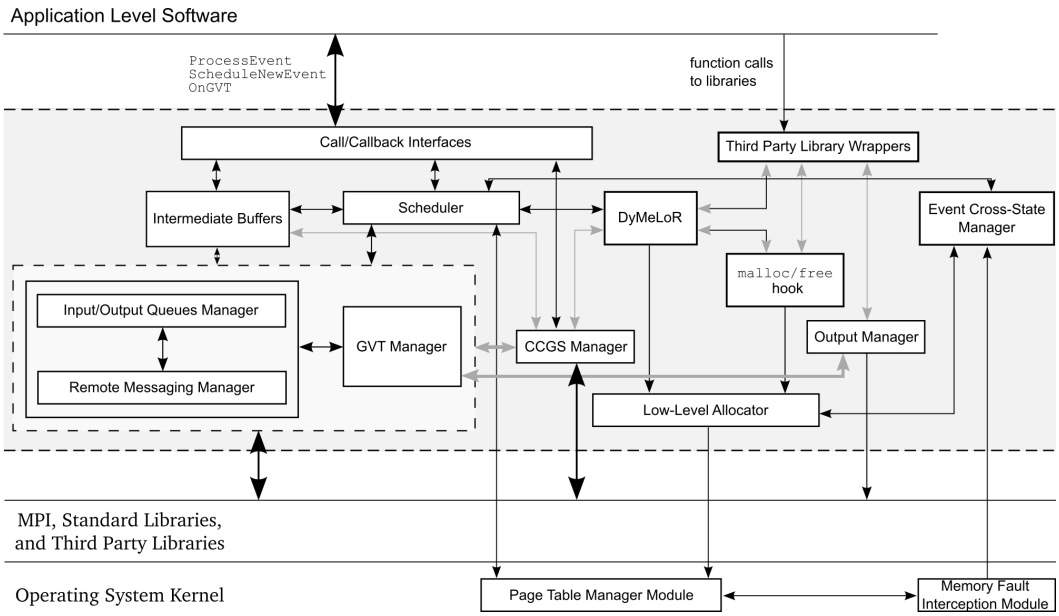

Figure 3.1. An overview of ROOT-Sim architecture

3.1 ROOT-Sim overview

ROOT-Sim is a PDES runtime environment developed in C/Posix and supports a transparent deploy on distributed clusters of massively-parallel compute nodes by relying on the MPI protocol, the HPC industry standard.

From the user point of view, ROOT-Sim presents itself as a wrapper for the gcc compiler, which links the model code with the ROOT-Sim runtime libraries as per user supplied options. The resulting executable can then run on a single node as well as on a parallel cluster.

ROOT-Sim architecture, depicted in Figure 3.1, has evolved in more than thirty years of development. It is quite involved, therefore this section describes only the high level interface. For a thorough description of the internals, we refer the reader to the project wiki at

`https://github.com/HPDCS/ROOT-Sim/wiki`

and the developer documentation at

`https://hpdcs.github.io/ROOT-Sim//docs/master/index.html`

3.1.1 ROOT-Sim high level interface

ROOT-Sim allows the model developer to use a simple application-callback function as the simulation event handler:

```
ProcessEvent(unsigned int lp_id, simtime_t now, unsigned int event_type,
void *event_payload, unsigned int size, void *state)
```

Its parameters determine when during the simulation time and which simulation object is currently taking control for processing its next event, and where the state of this object is located in memory.

The simulation model can generate a new event and inject it into the system, destined at any LP (even itself), by using:

```
ScheduleNewEvent(unsigned int receiver, simtime_t timestamp, unsigned
int event, void *event_payload, unsigned int size)
```

Its parameters specify which LP is the intended receiver of the event, and when in the future it has to process it. Clearly the supplied timestamp can't be smaller than the value of `now` passed to `ProcessEvent()`, as this would make the future affect the past, which is impossible.

The runtime environment enforces a (distributed) termination detection procedure by using the callback:

```
OnGVT(unsigned int lp_id, void *snapshot)
```

When the GVT is reduced, all LPs are asked whether the simulation (at that particular LP) can be considered as completed. In case that all LPs reply positively, the simulation is halted.

3.1.2 Interfacing with ROOT-Sim

ROOT-Sim provides a per-LP data structure, called `lp_struct` which it uses to provide stock PDES functionalities, such as storing snapshots and managing events queues. We extended it with the necessary memory pointers towards the data structures used to implement the API functionalities. We therefore

defined a new per-LP memory block which keep tracks of the topology state called `topology_t`, and another one to keep track of agents states and region neighborhoods called `region_t`.

To catch simulation events it is sufficient to override `ProcessEvent()` and `ScheduleNewEvent()` with our custom logic. The API is transparent to GVT computations, but our implementation attaches to the fossil collection in order to periodically cleanup its data structures. For performance reasons, our API runtime doesn't use the model memory manager for its allocations, instead it uses directly `malloc()` and `free()`. Rollbacks and restores of our data structures therefore must be handled manually when ROOT-Sim initiates a rollback or a restore of a LP.

We had to attach to ROOT-Sim argument parser to provide the model developer with a way to specify various parameters at model launch, such as for example, the path of the JSON topology file. ROOT-Sim argument parser is written using the GNU library argp.

3.2 Topology implementation details

This module provides ROOT-Sim with extended topology functionalities and it is implemented so as to be completely independent from the actual Agent-Based Modeling runtime. The JSON file containing the topology information is loaded at simulation startup; if it isn't supplied, default values are used. Weights are saved and processed as double-precision floating point numbers. We managed to fit the topology related state into a single contiguous memory block per LP. This renders rollbacks and restores of topology related information as simple as a single memory copy. The implementation deeply differentiates cost-based and probability-based topologies, for that reason we describe them separately.

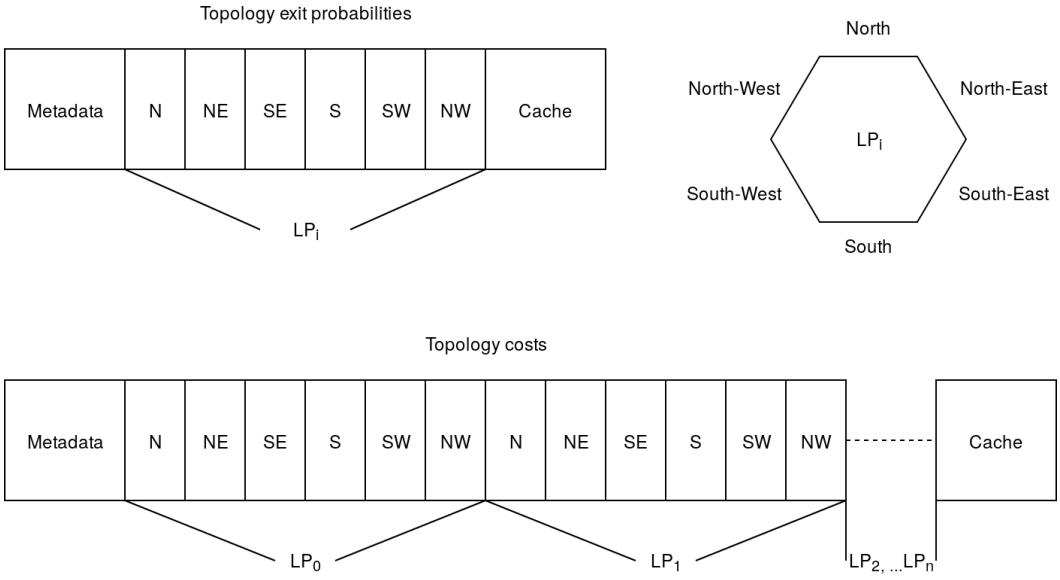

Figure 3.2. Comparison between cost-based and probability-based topologies memory layout

3.2.1 Cost-based topologies

A comparison of the memory layout of cost-based and probabilities based topologies is shown in Figure 3.2.

In a cost-based topology the whole cost matrix has to be stored by each LP in order to compute minimum cost paths correctly even in the frequent occurrence of rollbacks and LP logical times misalignments. In the worst case, the memory consumption scales as the cube of the number of LPs. Our implementation allows the model developer to mark his topology as read only; in this case memory consumption can be greatly reduced by storing only a single copy of the weights matrix per MPI kernel instance.

As mentioned before, our implementation uses Dijkstra's Shortest Path First algorithm to compute the minimum cost paths. Those are computed on demand when either `FindReceiverToward()` or `ComputeMinTour()` are called. Our Dijkstra's algorithm implementation uses a binary heap as priority queue achieving asymptotic time complexity of $O((num_L P + num_e dg))log_2(num_L P)$

where num_LP is the count of LPs while num_edg is the number of edges in the topology implied graph. In case we are dealing with a grid-base geometry E scales linearly with num_LP, a satisfying running time. In the worst case, such as dense graph-based topology geometries, num_edg is quadratic with respect to num_lp. We employ Neumaier's algorithm to compute the partial weights sums; this is needed in order to avoid the loss in precision deriving from repeated sums of floating point numbers.

We expect that the vast majority of minimum path computations would have source LP coincident with the LP which issued the call. For this reason, `topology_t` structures maintain a cache of the minimum spanning tree rooted in the related LP (the memory area is indicated in Figure 3.2). Clearly we need to invalidate that cache if some weights have changed value. With one single run of Dijkstra we can update the cache, this explains why we use this algorithm instead of a simpler Depth First Traversal. With the aid of this cache it is possible to serve in constant time `FindReceiverToward()` calls, and in linear time (instead that the log-quadratic worst case) calls to `ComputeMinTour()` with the "right" source.

Calls to the API `SetValueTopology()` are implemented with a special control event that is broadcast to all the LPs (except the one originating the request) with timestamp equal to the currently processed event. These control messages follow the same logic of the model's ones with respect to rollbacks and antimessages. The only difference is that they are hidden to the model developer; they are intercepted instead directly by our API runtime. Whenever such a control message is received, the API simply applies the requested weight to its local copy of the cost matrix.

3.2.2 Probability-based topologies

The `topology_t` data structure only needs to store weight values associated with the related LP. This leads to a consumption of memory which is quadratic

with respect to the number of LPs in the worst case. This doesn't sound satisfying, but it isn't possible to improve the situation significantly simply because the edge count of a graph is quadratic in the worst case.

Stochastic functionalities such as `FindReceiver()` are implemented with the aid of ROOT-Sim rollback-able PRNG library. The PRNG supports rollbacks in order to keep consistency even during silent execution of events. Let the weight associated with the exit probability toward LP_i be w_i. The naive approach for randomly selecting the exit LP would be:

1. compute $sum_w = \sum_{k=0}^{n} w_k$
2. select a random floating point number $r \in]0, sum_w]$
3. select LP_j as exit LP with $j = min\{i \mid r < \sum_{k=0}^{i} w_k\}$

If the number n of neighbors regions is high or if the weight values exhibit a large dynamic range, this approach misbehaves because of the limited precision of floating point numbers. This other algorithm is used instead:

1. compute $sum_w = \sum_{k=0}^{n} w_k$
2. select a random integer number $j \in [0, n-1]$
3. select a random floating point number $r \in]0, sum_w]$
4. select LP_j as exit LP if $r < w_j$, else restart from step 2

It can be shown that the latter procedure is mathematically equivalent to the former one, but for our application the last one is undoubtedly more accurate. The trade off is the chance of computing many iterations of the algorithm to conclude a random selection. To alleviate the impact of this algorithm, the value sum_w is cached, refreshing it whenever a local probability value is changed.

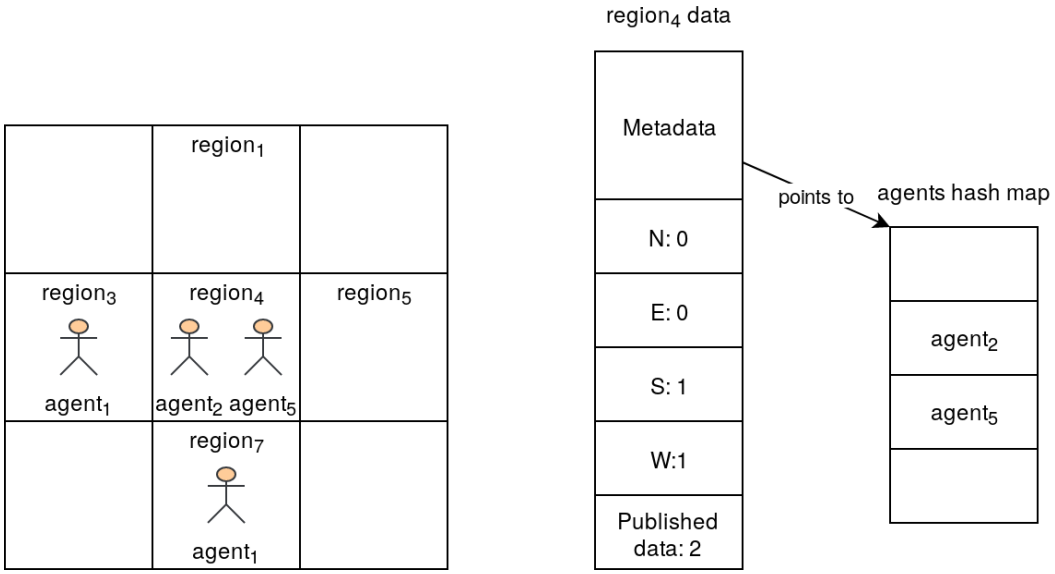

Figure 3.3. ABM runtime memory layout: `TrackNeighbourInfo()` used to keep count of adjacent agents

Calls to the API `SetValueTopology()` require a single message if the source LP of the request is different from the one issuing the call. If this isn't the case, the update request can be trivially handled locally.

3.3 Agent-Based Modeling runtime

The ABM module heavily depends on the topology runtime described earlier. In fact the initialization phase of this module needs topological information in order to set up correctly sized memory buffers. During operation, the ABM module needs the topology APIs in order to move the agents following minimum cost paths or respecting the probability distributions decided by the model developer. The `region_t` structure associated to each LP is fairly simple as it is a contiguous memory block. The most interesting piece is the carefully tuned agents hash map but we will describe it later.

The implementation of `TrackNeighborInfo()` consists in registering a pointer to the memory area inside the `region_t` structure. After each processed

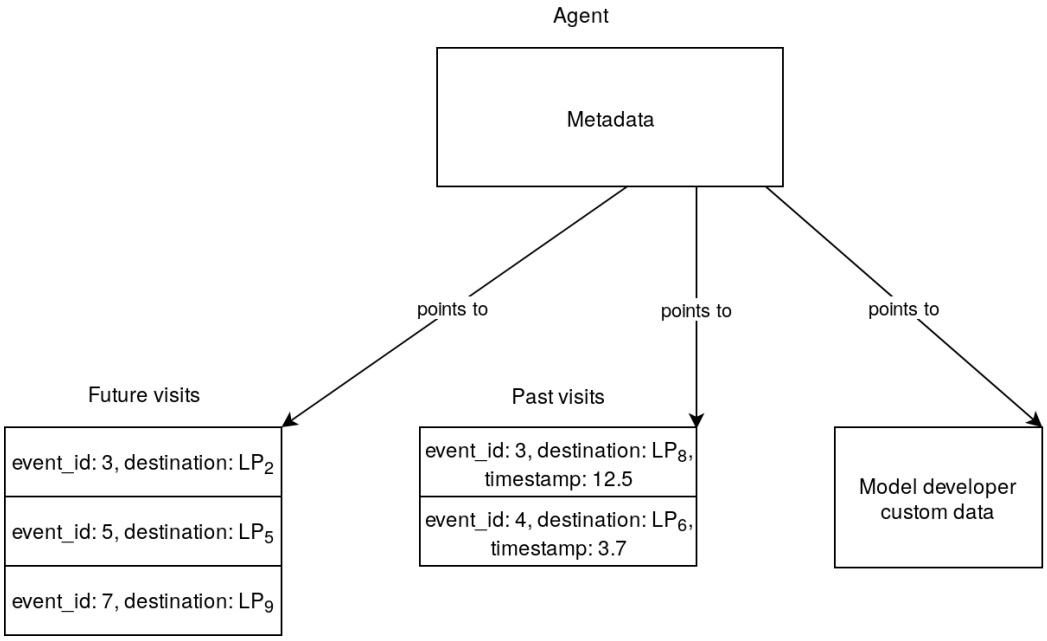

Figure 3.4. The agent memory layout as per our implementation

event, the runtime checks if the tracked area has changed and if it did, a special control event containing the new memory data is broadcast to every region's neighbor. Upon reception of such a control event the runtime writes the data in the corresponding memory area inside the `region_t` structure as shown in the example in Figure 3.3. This eager diffusion protocol is necessary in order to keep consistency but, as the assessment will show, it comes with a big performance cost: a rollback in a region may cause a chain of rollbacks in several LPs, effectively serializing the simulation execution.

The agent memory representation is depicted in Figure 3.4. Visits APIs require the agent to maintain two lists with the future and past regions visits. These are two separate list because a model may not need past visits functionalities; the model developer can disable past visits logging with a flag, potentially saving large quantities of memory. `GetVisit()`, `EnqueueVisit()` and the other related functions are therefore implemented as simple list operations.

`ScheduleNewLeaveEvent()` is implemented as a control message sent with the intended agent leave as timestamp to the very same LP issuing the call.

This control event is caught by our runtime which verifies that the agent is still alive or in the region. Then `ProcessEvent()` is called so that the model can be informed of the leaving agent. When an agent actually leaves (the model can still kill the agent when it is informed of his intention to leave), another type of control message containing the serialized agent is sent to the destination LP. At the same time the agent is removed from the hashmap. At this point the receiving LP only needs to deserialize the agent, insert it in the hashmap and then call `ProcessEvent()` to inform the model of the visit event.

Removal is done lazily by flagging the killed agents. During fossil collections the flagged agents are definitely cleaned up.

3.3.1 The agents data structure

In our API implementation, the `agent_t` which uniquely identifies the agents is simply a 64 bit number generated using the aforementioned *Cantor pairing function*. To support the retrieval of the agents state we need to store them in a dictionary-like data structure. In order to efficiently doing so, it is necessary to carefully choose a proper data structure. We opted for a hash table with linear probing through Robin Hood hashing and backward shift deletion[6].

Let's define dib_x as the distance between the position x would have in the empty table and its actual one (Distance from Initial Bucket). Insertion of a new element A using the Robin Hood hashing strategy works as linear probing with one important difference: instead of inserting A necessarily in the first empty slot, it is swapped with the currently probed table element B if $dib_B < dib_A$ (dib_A is computed using the position A would have if inserted in B's place). The insertion would then continue with B. The operation ends when an empty slot is found.

Backward shift deletion coincides with the traditional one followed by an additional step: all the elements positioned after the deleted one following the insertion probing direction, are shifted back by one slot. This process is ended

as soon as an element with *dib* equal to 0 is encountered. This reduces the *dib* of the shifted elements by one.

The effect of this policies is reducing the mean and standard deviation of *dib* which traduces in faster look-ups and insertion on average. The table size is constrained to be a power of two so resizes are performed doubling or halving the table size. This way, elements positions can be computed with a right shift of their hash. It has been shown experimentally[22] that even with load factors nearing 0.9, the mean probe sequence length is only seven. We therefore selected 0.85 as the upper load factor limit before a table resize is performed. To limit memory consumption during long simulation runs, we also resize the table if its load factor goes below 0.15. These precautions guarantee us constant amortized time insertion, lookup and deletions. Separate chaining achieves a comparable performance level but with one drawback which is major for our application: separate chaining handles a new collision allocating a new memory block necessary for the chaining list. This means that in average, the number of distinct memory blocks involved in a rollback would scale linearly with the number of elements contained in the data structure. For comparison, snapshots and restores of our hash table need only exactly two distinct memory copies.

Since our agents are identified by a 64 bit number, it is possible to select a very fast hash function by sacrificing the possibility to compute hashes of arbitrarily long data sequences. We adapted a variant of the SplitMix pseudo random number generator[52] to act as an hashing function. This solution yields good quality hashes with few shifts and multiplications.

Chapter 4

Assessment

4.1 Test-Bed Models and Environment

For our assessment, we consider for this study 3 synthetic models, and 3 real-world applications. The considered models are the following:

1. *Stupid Model* [43]: in a toroidal region, several bugs (the agents) move randomly. Each cell produces food, at a certain production rate, which is eaten by the bugs which enter it—while eating, a bug grows in size. Bugs can move to another cell only if the destination cell is empty, and they select the empty neighboring cell with the highest food amount. With a certain probability, when a bug enters a cell, it either dies or reproduces. When a bug reproduces, it spawns five new bugs in surrounding cells if they are empty. If not enough empty cells are available, the bug spawns a smaller number of children. The simulation halts when the largest bug reaches a certain size.

2. *Segregation* [47]: this social model is used to show how segregation among people happens, even though people (the agents) do not mind being surrounded by other agents of a different race. An agent is modeled by its race, ethnicity, economic status, etc. Multiple populations occupy a

certain environment with random positions. An agent can be satisfied or not with its current location, in the sense that an agent is surrounded by a certain percentage of agents that are like itself. In the negative case, the agent moves to a different (empty) location.

3. *Sugarscape* [15]: in a grid environment, every cell contains different amounts of sugar. The agents move around by targeting the cell filled with the largest amount of sugar. Once they eat, they metabolize and leave pollution. In a way similar to the stupid model, they can die and reproduce. Additionally, the agents can trade or borrow sugar, generate immunity or transmit diseases.

4. *Terrain-Covering Ant Robots* (TCAR) [28]: This is an agent-based model particularly interesting for the assessment of rescue scenarios. If some kind of accident occurs in a region which is either unknown by the rescuers or altered by the accident itself, the first action in order to actually rescue the victims is to explore the whole region. The terrain is modeled as an undirected graph, accounting also for obstacles, in which ant robots move in every direction. While moving, they leave *pheromones*, i.e. they implement a node-counting algorithm, where each cell is assigned a counter which gets incremented whenever any robot visits it. Each ant picks its next direction by selecting the neighboring cell with the smallest visit counter value.

5. *Robot Explore* [16]: A group of robots is set out into an unknown space, with the goal of exploring it. They keep a representation of the explored world, so as to determine which is the closest unexplored area they can reach, and they compute the shortest path to reach it. While moving around, the robots gather measurements of the environment and store them in their state. During the exploration, some accidents can make a region non-traversable. The robots explore independently, until one

coincidentally detects another robot in its proximity. Once two robots meet, they can exchange the information sensed from the environment and make collaborative exploration decisions.

6. *Tuberculosis* [35]: This model allows to simulate the spread of tuberculosis infections. It has been effectively used to study this phenomenon in the city of Barcelona. Individuals (the agents) can move around. They can be healthy, infected, sick, under treatment, or susceptible to relapse. Depending on their interactions, healthy individuals might become infected. Sick agents might be susceptible to death. This is an overall epidemiological model which requires a very large number of agents, with complex state transitions, to allow the identification of emergent behavior related to the spread of the disease.

In the experimental assessment, we report performance data of 4 of the aforementioned models. We have used a cluster of three medium size heterogeneous servers connected via a 1 GB dedicated Ethernet connection. Two servers are 32-core (AMD Opteron 6168) machines equipped with 64 GB of RAM. One is a 48-core (AMD Opteron 6168) machine equipped with 128 GB of RAM. All machines run Linux 4.9.0. We have used gcc 8.2.0 to compile the software, and OpenMPI 3.1.3 as the MPI runtime environment. In the experiments, the total number of threads has been varied from 3 to 112. The MPI assignment policy has been set so as to ensure that all three machines are used, by distributing the threads on the cluster in a round-robin fashion. This is a worst-case scenario, especially when the number of threads is low, because we always incur in the communication cost even though a single machine could support message passing by relying on shared memory.

Model	Original Implementation	Using our API
Stupid Model	189	138
Segregation	83	110
Sugarscape	1072	152
TCAR	228 [1]	103
Robot Explore	500	332
Tuberculosis	1,115	654

Table 4.1. LOCs for the different Model Implementations

4.2 Effects on Programmability

While assessing the effects on programmability is not easy, especially in contexts which involve models coming from multiple domains, we are taking into account here classical agent-based models from the literature, and we compare their classical implementations with ones made relying on the proposed API. For each of the aforementioned models, we have considered the original implementation (where available). We have compared the Lines of Code (LOCs) of the original implementation with the LOCs of our re-implementations using the discussed API.

Table 4.1 shows the comparison. As it can be seen, except for segregation model, the line count is always smaller when implementing the models by relying on the proposed API. It should be kept in mind that segregation model original implementation It is interesting to note that this is the case also when dealing with very simple (toy) models, such as the Stupid Model. This is an indication of the fact that the API is effective at capturing the needs of ABM, thus reducing the burden on the model developer which can benefit

[1]The original paper in [28] does not provide a reference implementation. Therefore, we report here the LOCs associated with a re-implementation of the model, according to the specification, which does not rely on our API.

Model	# LPs	Involved Agents
Stupid Model	4,096	1 [2]
Segregation	10,000	10,000
TCAR	4096	16
Tuberculosis	1024	400,000

Table 4.2. Configurations of the models

from the performance speedup and the increased amount of working memory traditionally offered by distributed PDES.

4.3 Performance Assessment

We present performance data of 4 of the aforementioned simulation models on the described hardware infrastructure. The models used and their configurations are reported in Table 4.2. To show the viability of our proposal from a performance point of view, we report the speedup of the distributed runs over a sequential simulation. Sequential simulations are extremely optimized, as they are based on a fast O(1) scheduler based on a calendar queue [4]. We remind that the experimental setup is a worst case, due to the fact that we always incur in increased network activity when we rely on a reduced number of threads, due to the distribution of the threads in the cluster.

In Figure 4.1 we report the performance data related to the synthetic benchmarks Stupid Model and Segregation. It is interesting to note that both models exhibit a linear speedup on the machine, despite the fact that the event granularity is quite small (on the order of tens of microseconds) and the amount of events exchanged across the LPs is non-minimal—this latter point is related to the fact that in both models we rely on the `TrackNeighbourInfo()` API which transparently sends control messages to neighboring LPs upon any

[2]This is the initial agent count. During the simulation, the number of agents increases.

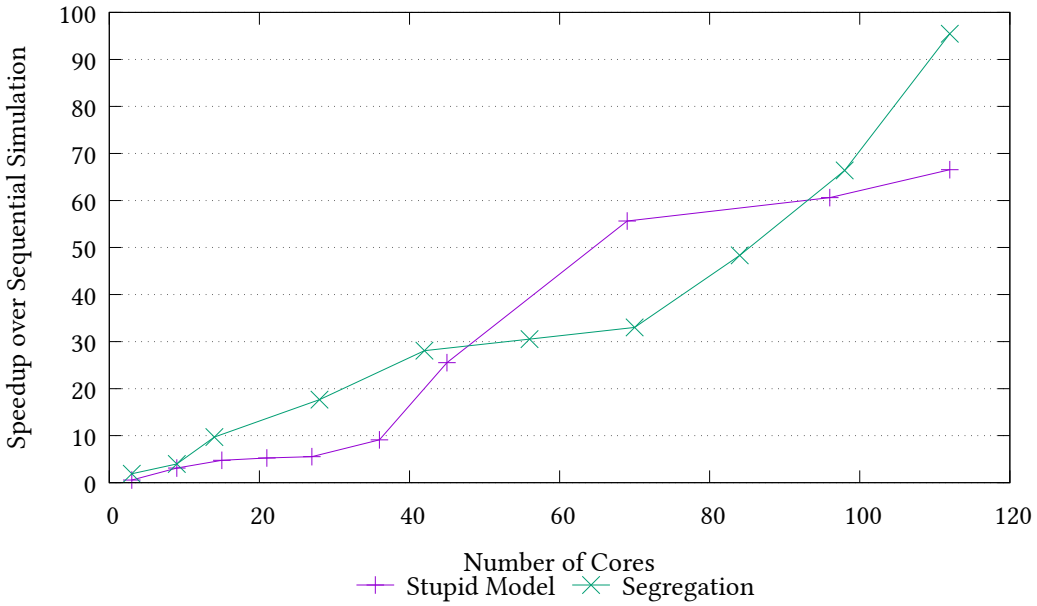

Figure 4.1. Results with Synthetic Models.

update, and the number of agents in the model (especially in the segregation case) is quite high. The reason behind this point is that the scale of the model can be significantly increased (up to 10,000 LPs in the Segregation model case) due to the fact that simulations states are very small, and so is the number of explanatory variables used to describe an agent. Anyhow, this result suggests that if these models are used (as they are intended to) as skeletons to build more complicated logic, then these models will benefit from the increased computing power offered by distributed PDES.

In Figure 4.2 we report the performance results for two real-world simulation models, namely tuberculosis (TBC in the plot) and TCAR. With respect to TCAR, we can see that after a certain number of concurrent threads, the performance observes a significant drop. This is a foreseeable results, taking into account the configuration reported in Table 4.2. Indeed, there is a high imbalance between the number of active agents and the regions in the model. Given the distributed nature of the simulation, the probability that LPs observe a skew in the simulation time is high. This, in its turn, affects

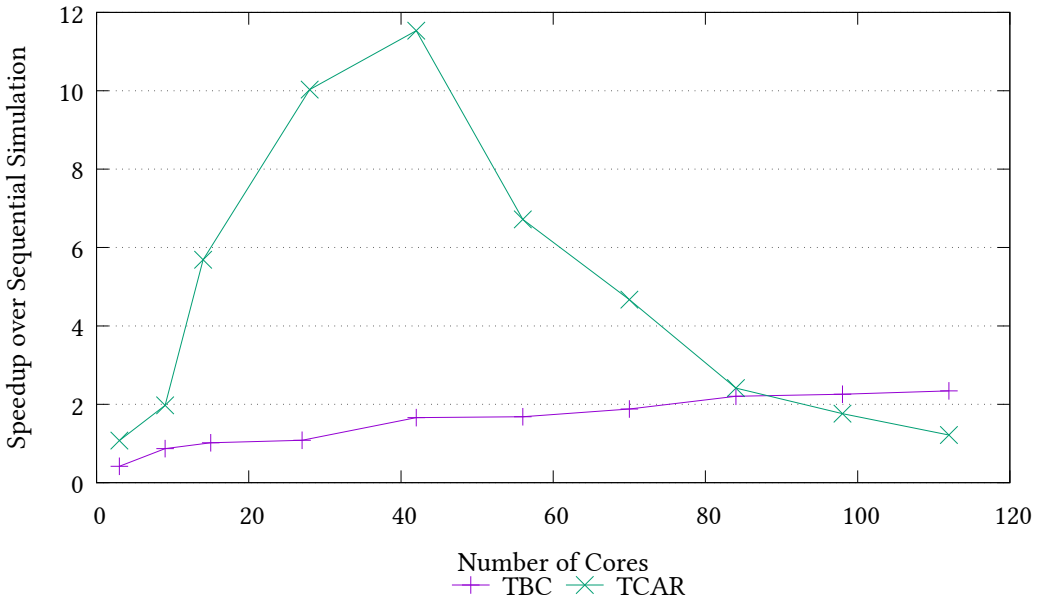

Figure 4.2. Results with Real-World Applications.

the overall efficiency of the simulation—in our runs, it got as low as 15% in some configurations. While this is a phenomenon which would have appeared independently of the API used to implement the model, we can note that the proposed API and its implementation is resilient to this unfavorable scenario up to a non-negligible number of distributed thread (42).

With respect to TBC, we note that this is the model with the highest degree of parallelism, due to the fact that one thread manages a reduced number of LPs (9 LPs per thread is the minimum). This is a scenario which is known to increase the rollback probability. Moreover, this is also exacerbated by the fact that the number of agents in the simulation, compared to the size of the environment, is quite large. The proposed API, in this case, is able to provide anyhow a speedup, although minimal. It is important to note that this is a clear case in which resorting to our proposal allows to make unfeasible problems feasible, due to the possibility to overcome the memory wall thanks to the increased number of nodes used for the simulation. In this specific model, the sequential simulation has shown a memory footprint of more than

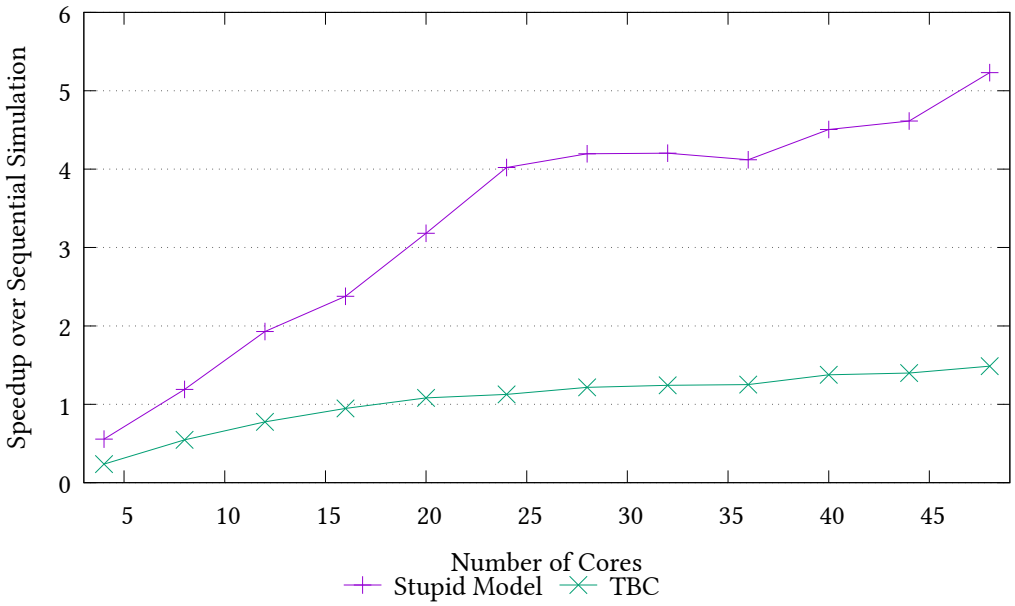

Figure 4.3. Results using a single node.

16 GB, which could not be manageable when using off-the-shelf commodity hardware.

For the sake of completeness, in Figure 4.3 we report performance data taken when running on the largest single node of our cluster, fo the real-world applications. Here, we do not pay communication costs, but the trends of the curves are perfectly similar to the ones shown in Figure 4.2, thus showing that the proposed API and its reference implementation is also resilient to network delays.

Chapter 5

Conclusions and Future Work

In this thesis we have introduced an API specification for ABM in speculative runtime environments implemented according to the Time Warp synchronization protocol. This API has been shown to be effective at implementing very different agent-based models in a compact and expressive way. An implementation of our API has also shown that it is possible to obtain non-minimal speedup also in very simple (toy) models. We therefore consider this as an important step ahead at disclosing the power of PDES to domain expert which should not be exposed to the complexity of speculative synchronization.

We deem community feedback fundamental for the reason that a programming interface is useful only if it is proficiently used. We recognize that the APIs need a refinement in order to provide a better development experience. The implementation needs to improve as well: while simpler models showed very good scaling capabilities, the *Tuberculosis* model doesn't scale well enough on distributed architectures using our runtime implementation. We nonetheless feel that is a very promising work, therefore working on the worst bottleneck, the `TrackNeighbourInfo()` and `GetNeighbourInfo()` API implementations. We will also try to port the API to a higher level language such as Python; we recognize that C development can be a very daunting task for the non IT-professional.

The implied purpose of this work is to initiate an interaction between us in the HPC community, and the domain experts, in order to reach an agreement on the set of needed ABM runtime features. This would allow on one side, to focus on the optimization of a clear set of functionalities while offering, on the other side, a standardized interface to the model developers.

Bibliography

[1] Abar, S., Theodoropoulos, G. K., Lemarinier, P., and O'Hare, G. M. Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review*, **24** (2017), 13. Available from: https://linkinghub.elsevier.com/retrieve/pii/S1574013716301198, arXiv:arXiv:1011.1669v3, doi:10.1016/j.cosrev.2017.03.001.

[2] Allen, T. T. *Introduction to Discrete Event Simulation and Agent-based Modeling*. Springer London, London (2011). ISBN 978-0-85729-138-7. doi:10.1007/978-0-85729-139-4.

[3] Bonabeau, E. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, (2002). arXiv:1709.03423, doi:10.1073/pnas.082080899.

[4] Brown, R. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, **31** (1988), 1220.

[5] Carothers, C. D., Perumalla, K. S., and Fujimoto, R. M. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, **9** (1999), 224.

[6] Celis, P., Larson, P.-A., and Munro, J. I. Robin hood hashing. In *Proceedings of the 26th Annual Symposium on Foundations of Computer*

Science, SFCS '85, pp. 281–288. IEEE Computer Society, Washington, DC, USA (1985). ISBN 0-8186-0844-4. Available from: https://doi.org/10.1109/SFCS.1985.48, doi:10.1109/SFCS.1985.48.

[7] Cingolani, D., Pellegrini, A., and Quaglia, F. RAMSES: Reversibility-based agent modeling and simulation environment with speculation support. In *Proceedings of Euro-Par 2015: Parallel Processing Workshops* (edited by S. Hunold, A. Costan, D. Ginenéz, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander), PADABS, pp. 466–478. LNCS, Springer-Verlag (2015). doi:10.1007/978-3-319-27308-2_38.

[8] Cingolani, D., Pellegrini, A., and Quaglia, F. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. *ACM Transactions on Modeling and Computer Simulation*, **27** (2017), 1. doi:10.1145/3077583.

[9] Collet, Y. xxHash: Extremely Fast Hash Algorithm (2015). Available from: http://www.xxhash.com/.

[10] Collier, N. and North, M. Parallel agent-based simulation with Repast for High Performance Computing. *SIMULATION*, **89** (2013), 1215. doi:10.1177/0037549712462620.

[11] Colman, A. M. The complexity of cooperation: Agent-based models of competition and collaboration. *Complexity*, (1998).

[12] Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., and Spagnuolo, C. A framework for distributing agent-based simulations. In *Proceedings of Euro-Par 2011: Parallel Processing Workshops* (edited by M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. L. Scott, J. L. Traff, G. Vallée, and J. Wei-

dendorfer), Lecture Notes in Computer Science, pp. 460–470. Springer Berlin Heidelberg (2012). ISBN 978-3-642-29736-6. Available from: http://link.springer.com/10.1007/978-3-642-29737-3{_}51, doi:10.1007/978-3-642-29737-3_51.

[13] Dickman, T., Gupta, S., and Wilsey, P. A. Event pool structures for PDES on many-core Beowulf clusters. In *Proceedings of the 2013 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 103–114. ACM Press (2013). ISBN 9781450319201. doi:10.1145/2486092.2486106.

[14] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, **1** (1959), 269. doi:10.1007/BF01386390.

[15] Epstein, J. and Axtell, R. L. *Growing Artificial Societies: Social Science from the Bottom Up* (1997). ISBN 9780874216561. arXiv:9809069v1, doi:10.1007/s13398-014-0173-7.2.

[16] Fox, D., Ko, J., Konolige, K., Limketkai, B., Schulz, D., and Stewart, B. Distributed Multirobot Exploration and Mapping. *Proceedings of the IEEE*, **94** (2006), 1325. doi:10.1109/JPROC.2006.876927.

[17] Fujimoto, R. M. Parallel Discrete Event Simulation. *Communications of the ACM*, **33** (1990), 30.

[18] Fujimoto, R. M. and Hybinette, M. Computing Global Virtual Time in Shared-Memory Multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, **7** (1997), 425. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.5956, doi:10.1145/268403.268404.

[19] Gardner, M. Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, **223** (1970), 120.

[20] Gilbert, N. and Terna, P. How to build and use agent-based models in social science. *Mind & Society*, **1** (2000), 57. Available from: http://link.springer.com/10.1007/BF02512229, doi:10.1007/BF02512229.

[21] Goldsman, D., Nance, R., and Wilson, J. A brief history of simulation. pp. 310 – 313 (2010). doi:10.1109/WSC.2009.5429341.

[22] Goossaert, E. Robin hood hashing: backward shift deletion (2013). Available from: http://codecapsule.com/2013/11/17/robin-hood-hashing-backward-shift-deletion/.

[23] Holcombe, M., Coakley, S., and Smallwood, R. A general framework for agent-based modelling of complex systems. In *Proceedings of the 2006 European conference on complex systems*. European Complex Systems Society Paris, France (2006).

[24] Iba, H. *Agent-Based Modeling and Simulation with Swarm*. Chapman and Hall/CRC (2013). ISBN 9781466562400. doi:10.1201/b15024.

[25] Jefferson, D. R. Virtual Time. *ACM Transactions on Programming Languages and System*, **7** (1985), 404.

[26] Jefferson, D. R. Virtual time II: storage management in conservative and optimistic systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pp. 75–89. ACM, New York, NY, USA (1990). ISBN 0-89791-404-X. Available from: http://doi.acm.org/10.1145/93385.93403, doi:10.1145/93385.93403.

[27] Karpov, Y. G. AnyLogic — a New Generation Professional Simulation Tool. In *Proceedings of the 6th International Congress on Mathematical Modeling*, MATHMOD (2004).

[28] Koenig, S. and Liu, Y. Terrain Coverage with Ant Robots: a Simulation Study. In *Proceedings of the fifth international conference on Autonomous*

agents, AGENTS, pp. 600–607. ACM (2001). ISBN 1-58113-326-X. doi:10.1145/375735.376463.

[29] LaPre, J. M., Gonsiorowski, E. J., and Carothers, C. D. LORAIN: a step closer to the PDES 'holy grail'. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of Advanced Discrete Simulation*, PADS, pp. 3–14. ACM Press, New York, New York, USA (2014). ISBN 9781450327947. doi:10.1145/2601381.2601397.

[30] Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. MASON: A multiagent simulation environment. *Simulation*, **81** (2005), 517. doi:10.1177/0037549705058073.

[31] Mabu, S., Hirasawa, K., and Hu, J. A Graph-based Evolutionary Algorithm: Genetic Network Programming (GNP) and its Extension sing Reinforcement Learning. *Evolutionary computation*, **15** (2007), 369. Available from: http://www.ncbi.nlm.nih.gov/pubmed/17705783, doi:10.1162/evco.2007.15.3.369.

[32] Macal, C. M. and North, M. J. Tutorial on Agent-Baesd Modeling and Simulation. In *Proceedings of 2005 Winter Simulation Conference* (2005). ISBN 9788578110796. arXiv:arXiv:1011.1669v3, doi:10.1017/CBO9781107415324.004.

[33] McKee, S. a. Reflections on the memory wall. In *Proceedings of the first conference on Computing Frontiers*, p. 162. ACM Press, New York, New York, USA (2004). ISBN 1581137419. Available from: http://portal.acm.org/citation.cfm?doid=977091.977115, doi:10.1145/977091.977115.

[34] Mithen, S. Stepping out: a computer simulation of hominid dispersal from Africa. *Journal of Human Evolution*, **43** (2002),

433. Available from: http://linkinghub.elsevier.com/retrieve/pii/S0047248402905841, doi:10.1016/S0047-2484(02)90584-1.

[35] Montañola-Sales, C., Gilabert-Navarro, J.-F., Casanovas-Garcia, J., Prats Soler, C., López Codina, D., Ribas Valls, J., Cardona Iglesias, P. J., and Vilaplana, C. Modeling tuberculosis in Barcelona. A solution to speed-up agent-based simulations. In *Proceedings of the 2015 Winter Simulation Conference*, pp. 1295—-1306. IEEE Computer Society (2015).

[36] Montañola-Sales, C., Onggo, B. S., Casanovas-Garcia, J., Cela-Espín, J. M., and Kaplan-Marcusán, A. Approaching parallel computing to simulating population dynamics in demography. *Parallel Computing*, (2016). doi:10.1016/j.parco.2016.07.001.

[37] MPI Forum. Message Passing Interface Forum. http://www.mpi-forum.org/ (1994). Available from: http://www.mpi-forum.org/.

[38] Pellegrini, A. and Quaglia, F. The ROme OpTimistic Simulator: A tutorial. In *Proceedings of the Euro-Par 2013: Parallel Processing Workshops* (edited by D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Constan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer), PADABS, pp. 501–512. LNCS, Springer-Verlag (2014). Available from: http://link.springer.com/10.1007/978-3-642-54420-0{_}49, doi:10.1007/978-3-642-54420-0_49.

[39] Pellegrini, A. and Quaglia, F. Wait-free Global Virtual Time computation in shared memory Time-Warp systems. In *Proceedings of the 26th International Conference on Computer Architecture and High Performance Computing*, SBAC-PAD, pp. 9–16. IEEE Computer Society, Paris, France (2014). ISBN 978-1-4799-6905-0. Avail-

able from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6970641, doi:10.1109/SBAC-PAD.2014.38.

[40] Pellegrini, A., Quaglia, F., Montanola-Sales, C., and Casanovas-Garca, J. Programming agent-based demographic models with cross-state and message-exchange dependencies: A study with speculative PDES and automatic load-sharing. In *Proceedings of the 2016 Winter Simulation Conference*, WSC, pp. 955–966. IEEE (2016). ISBN 978-1-5090-4486-3. doi:10.1109/WSC.2016.7822156.

[41] Pellegrini, A., Vitali, R., and Quaglia, F. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS*, pp. 45–53. IEEE (2009). ISBN 9780769537139. doi:10.1109/PADS.2009.24.

[42] Pugh, W. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, **33** (1990), 668. Available from: http://portal.acm.org/citation.cfm?doid=78973.78977, doi:10.1145/78973.78977.

[43] Railsback, S. F., Lytinen, S., and Grimm, V. StupidModel and Extensions: A Template and Teaching Tool for Agent-based Modeling. Tech. rep., Swarm Development Group (2005).

[44] Reynolds, C. W. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, **21** (1987), 25. doi:10.1145/37402.37406.

[45] Richmond, P. and Romano, D. Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu. In *Proceedings International Workshop on Supervisualisation* (2008).

[46] Rubio Campillo, X., Cela, J. M., and Hernàndez Cardona, F. X. Simulating archaeologists? Using agent-based modelling to improve battlefield excavations. *Journal of Archaeological Science*, **39** (2012), 347. doi:10.1016/j.jas.2011.09.020.

[47] Schelling, T. C. Sorting and Mixing. In *Micromotives and Mmacrobehavior*, chap. 4, p. 270. W W Norton & Co (1978). ISBN 02549948.

[48] Sleator, D. D. and Tarjan, R. E. Self-adjusting binary search trees. *Journal of the ACM*, **32** (1985), 652. Available from: http://portal.acm.org/citation.cfm?doid=3828.3835, doi:10.1145/3828.3835.

[49] Tisue, S. and Wilensky, U. Netlogo: A simple environment for modeling complexity. In *Proceedings of the International Conference on Complex Systems*, ICCS, pp. 1–10. NECSI (2004). ISBN 0769520723. Available from: http://ccl.sesp.northwestern.edu/papers/netlogo-iccs2004.pdf.

[50] Toccaceli, R. and Quaglia, F. DyMeLoR: Dynamic Memory Logger and Restorer Library for Optimistic Simulation Objects with Generic Memory Layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pp. 163–172. IEEE Computer Society (2008). ISBN 978-0-7695-3159-5. doi:http://dx.doi.org/10.1109/PADS.2008.23.

[51] Tocci, T., Pellegrini, A., Quaglia, F., Casanovas-García, J., and Suzumura, T. ORCHESTRA: An asynchronous wait-free distributed GVT algorithm. In *Proceedings of the 2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications*, DS-RT (2017). ISBN 9781538640289. doi:10.1109/DISTRA.2017.8167666.

[52] Vigna, S. Further scramblings of marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*, **315** (2017), 175 . Available from: http://www.sciencedirect.com/science/article/pii/S0377042716305301, doi:https://doi.org/10.1016/j.cam.2016.11.006.

[53] Vitali, R., Pellegrini, A., and Quaglia, F. Towards Symmetric Multi-threaded Optimistic Simulation Kernels. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS, pp. 211–220. IEEE (2012). ISBN 978-1-4673-1797-9. Available from: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6305914http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6305914http://ieeexplore.ieee.org/document/6305914/, doi:10.1109/PADS.2012.46.

[54] Vulov, G., Hou, C., Vuduc, R., Fujimoto, R., Quinlan, D., and Jefferson, D. The Backstroke framework for source level reverse computation applied to parallel discrete event simulation. In *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pp. 2960–2974. IEEE (2011). ISBN 978-1-4577-2109-0. doi:10.1109/WSC.2011.6147998.

[55] Wittek, P. and Rubio-Campillo, X. Scalable agent-based modelling with cloud HPC resources for social simulations. In *Proceedings of the 4th International Conference on Cloud Computing Technology and Science*, CloudCom, pp. 355–362. IEEE Computer Society (2012). doi:10.1109/CloudCom.2012.6427498.

[56] Xiao, J., Andelfinger, P., Eckhoff, D., Cai, W., and Knoll, A. Exploring Execution Schemes for Agent-Based Traffic Simulation on Heterogeneous Hardware. In *Proceedings of the 22nd International Symposium on Distributed Simulation and Real Time Applications*, DS-

RT, pp. 1–10. IEEE Computer Society (2018). ISBN 978-1-5386-5048-6. doi:10.1109/DISTRA.2018.8601016.