Data Synchronization in Distributed Simulation of Multi-Agent Systems

Paul Breugnot Bénédicte Herrmann Christophe Lang
Laurent Philippe

FEMTO-ST institute, Univ. Bourgogne Franche-Comté, CNRS, Besançon, France

Abstract

Modern Multi-Agent System simulations may involve millions of agents that are simulated over an extended period of time in order to better catch real world emergent properties. In this context, the usage of distributed computing resources may raise single machine limits both in terms of available memory and execution time. Distributing a simulation however implies lots of complex and specific issues as the data synchronization issues that we tackle here. Based on an interface that allows to develop models independently of the distribution, we propose the definition of *synchronization modes*, some inspired from existing platforms, other providing new features such as remote interactions. Since each mode comes with its pros and cons, guidelines are provided to help developers to find the best compromise for the distributed implementation of a model or a simulation platform. The performance of each mode is discussed and evaluated using a classical epidemiological *SIR* model.

Keywords: Multi Agent System Distributed Simulation High Performance Computing Data Synchronization

1 Introduction

Multi-Agent System (MAS) simulation is used in various fields such as biology, epidemiology, economics, sociology, energy management or traffic simulation. In any case, simulating real world phenomena might require the microscopic simulation of millions of agents on a large time scale: for example, EURACE [DvD08] aims at simulating the economic system at the scale of Europe, and the purpose of the ChiSIM [MCO$^+$18] model is to simulate the propagation of an epidemic in the city of Chicago at the individual scale. Such large scale simulations usually cannot be handled on a single machine, due to available memory and execution time limitations. The distributed execution on High Performance Computing (HPC) resources can be a solution to raise those limits since the intrinsic parallelism of MAS makes them good candidates to benefit from such architectures. On the other hand, the requirement for numerous and stochastic interactions among agents or with the environment, that become remote when agents are distributed, raises numerous issues since agents do not have a direct access to each others' memory: the only way to distribute information is through explicit message exchanges. Solving those issues in the context of MAS simulation is especially difficult considering the fact that MAS modelers are not expected to be HPC experts.

Several platforms, such as Repast HPC [CN12], D-MASON [CSS16] or FLAME [CWG$^+$12], among others [Rub14, BGLS17, BT19, STL13], provide solutions to issues related to agent modeling and execution on distributed platforms, in terms of communication scheme, load balancing or time synchronization. They however lack of data synchronization features to correctly manage remote interactions and the user is generally required to adapt his model to the platform, sometimes preventing the simulation of models with strong or specific constraints. The exchange of data with remote agents is indeed a critical feature that need to be solved efficiently in order to find a compromise between implementation complexity, performance, and model constraints.

This paper brings the following contributions:

- a formal definition of the data synchronization problem, with the proposition of a distribution independent interface and a generic and extensible specification of a synchronization mode concept.
- the definition and implementation of synchronization modes, unifying choices of other platforms, and introducing modes allowing remote modifications.
- a theoretical analysis of each mode, in terms of interaction rules and reproducibility levels, and a performance analysis based on an SIR model.

Following this introduction, data synchronization issues and their management within existing platforms are presented in the next section. The definition of synchronization modes and their analysis is then provided, and finally the performances of each mode are assessed using an SIR epidemiological model.

2 Data Synchronization in Distributed MAS Simulations

In a MAS simulation, allowed interactions, message exchanges or execution policies usually depend on the model definition [MS12]. More generally, MAS modeling techniques, supported by the simulators (NetLogo, Repast, MASON, GAMA...) imply constraints on the simulated models. Because of the distributed memory issues, distributed simulators usually constrain even more the models.

On HPC systems, the distribution of a MAS simulation consists in assigning agents and environment parts to a set of processes running on different nodes. This assignment is called *agent partitioning*. We define agents assigned to a process as *local* agents from this process. In order to allow remote interactions, *local* agents can access representations of agents executed on remote processes, that we call *distant* agents. Representations of *distant* agents are generally built according to the *local* agents perception fields, that can be considered as a geographical area of interest, or as neighbors in a graph.

In [BHLP21] authors show that different models might have different needs in terms of synchronization policy to properly run on a distributed architecture. For example, agents in a Flocking model only need to read their neighbors' positions. In consequence, it might be satisfying to read *local* agents' positions in place, and read *distant* agents' positions from a copy imported at the end of each time step. Another case is the Prey-Predator model where a predator agent might try to eat a *distant* agent, i.e. a prey executed on another process. According to the model rules, it means that (i) the prey state must be changed on the prey's origin process and (ii) it must be ensured that only one predator on all the processes will be allowed to eat this prey. In consequence, it is necessary to concurrently manage distant modifications within a time step.

Existing generic platforms supporting the distribution of models however lack support for different needs. In Repast HPC [CN12], *distant* agents are updated from their origin process at the end of each time step. The imported data overrides any modification performed on a *distant* agent during the time step. *Local* agents are directly accessed, and can be modified. D-MASON [CSS16] also updates *distant* agents' states at the end of each time step. But, in order to improve reproducibility, agents read *local* agents' data from the previous time step, so that data access does not depend on the *local* or *distant* state of agents. This might however imply severe limitations and the impossibility to simulate some models. The FLAME [CWG$^+$12] platform only allows agents to communicate through a common *message board*, updated at the end of each time step. This facilitates model distribution, since agents cannot directly access other agents' data, but concurrent modifications within the current time step are not allowed.

Other works [SDHP10, PRH$^+$03] propose the definition of methods where agents send action requests to a *conflict resolver*, that manages concurrency by sending back action results once all requests have been received. Being able to react to negative responses is however specific to some models. It fits in particular the Influence-Reaction [Mic07] modeling technique. Since the proposed implementations require global synchronizations before conflict resolutions, to ensure all requests are received for each agent, each agent must however wait for the execution of all others to know the result of its request, which is costly.

3 Synchronization Modes

Even if existing platforms define synchronization techniques, motivations are not always clear and proposed solutions might not fit the requirements of all models. It is then the responsibility of the user to adapt his model to the platform. For this reason we propose the synchronization mode concept based on a generic data synchronization interface.

In the following we first characterize agent interactions, then we describe the proposed interface that we use to specify synchronization modes.

3.1 Read and Write Operations

In order to provide a generic and meaningful interface to MAS modelers, describing agent interactions with *read* and *write* operations notably seems to satisfy most model requirements, even if behaviors are not directly described as such. For example, sending a message can be seen as a write operation in a buffer from the sender, and a read operation in the buffer from the receiver. In addition, the implementation of data synchronization in terms of *read* and *write* allows us to rely on existing parallel and distributed data management algorithms.

Table 1: Temporal Aspect of Interactions for Different Synchronization Modes

	self	*local*		*distant*	
Synchronization Mode	write	read	write	read	write
GhostMode	T	T	T	$T-1$	$\times$
GlobalGhostMode	$T+1$	$T-1$	$\times$	$T-1$	$\times$
HardSyncMode	T	T	T	T	T
PushGhostMode	T	T	T	$T-1$	$T+1$
PushGlobalGhostMode	$T+1$	$T-1$	$T+1$	$T-1$	$T+1$

It can then be observed that a fundamental aspect of synchronization is the temporal aspect of *read/write* operations. An agent executed at time step T can read data from time steps T or $T-1$, and write data to time steps T or $T+1$. A *read* at time $T-1$ corresponds to an access to a *ghost* copy, while a *read* at time T means that modifications performed by *writes* at time T will be perceived. A *write* at time T means other agents performing *reads* at time T will be able to perceive the modifications during the current time step, while a *write* at time $T+1$ is only accessible at the next time step. *Write* operations can also be prohibited in some cases.

Note that each operation has to specify the target agent with which an agent interact: (i) *self*: the target agent is the agent itself, (ii) *local*: the target agent is executed on the same process as the executed agent, (iii) *distant*: the target agent is executed on another process than the executed agent.

Then, using these observations, we can illustrate the concept of synchronization modes with several propositions that support different model specificities. In the *GhostMode*, which corresponds to the RepastHPC interaction implementation, the synchronization consists in allowing *writes* only on *local* agents while *distant* agents are *read* from a *ghost* copy of the system state at the previous time step ($T-1$). In the *GlobalGhostMode*, which corresponds to the D-MASON implementation of interactions, *local* and *distant* agents must always read other agents' data from a *ghost* copy. Finally, we define the *HardSyncMode* that allows concurrent and remote *reads* and *writes*, the strongest synchronization requirement.

Considering this, the previously introduced Flocking model might be simulated using *GhostMode* or *GlobalGhostMode*, while the *PreyPredator* model should be simulated using the *HardSyncMode*. Table 1 summarizes the temporal aspects of the proposed synchronization modes and others that will be discussed later in this section.

3.2 Data Synchronization Interface

Implementing data synchronization may turn out to be difficult for a model developer. For this reason, we propose the definition of a data synchronization interface, independently of the platform specification, in order to:

1. Propose a generic interface to implement models, independently of distribution or synchronization requirements.
2. Define a common specification, to characterize existing synchronization

modes and to define new ones. This allows to theoretically compare the properties of each mode, and to provide meaningful benchmarks.

This interface is based on five functions, inspired from lock mechanisms: `read(Agent agent)`, `release_read(Agent agent)`, `acquire(Agent agent)`, `release_acquire(Agent agent)` and `synchronize(Model model)`.

They allow to define protected blocks of code in which it is safe to call any existing agent method, rather that defining atomic *read/write* operations for each piece of data an agent can possibly own. In consequence, the only adaptations required from the user is to wrap instructions that access or modify agent data with the following guards:

`read(Agent agent), release_read(Agent agent)` : Any data can be safely *read* from the specified agent between `read()` and `release_read()` calls. How the internal data is updated (or not), i.e. which data is actually read, depends on the implemented synchronization mode.

`acquire(Agent agent), release_acquire(Agent agent)` : The `acquire()` operation gives to the calling process an exclusive access to the specified agent, so that any modification, considered as *write* operations, can be performed on the agent until `release_acquire()` is called. The actual behavior of those methods is also implementation defined, and does not require to take modifications into account.

`synchronize(Model model)` : This method is called from each process once it has finished to execute its *local* agents at the end of each time step.

Using this platform independent interface we can specify the implementation of synchronization modes. The implementation of each method then defines the properties supported by a synchronization mode, but must be provided in any case, even if some methods have no effect. Using this generic interface allows to develop a model without altering its implementation with synchronization dependent code and to transparently apply different modes on the same model.

3.3 Specification of Proposed Modes

Thus we provide some synchronization mode specifications that platform developers might implement according to their needs, independently of the simulation environment. This requires that some predefined distribution specific features, not discussed in this work, are provided:

1. Methods to query the neighbors of each *local* `Agent`. The neighbors might be *local* or *distant*, so that the agents' neighborhoods are properly preserved [BHLP21]. The neighborhood relation is model dependent (neighbors in a graph, agents in the Moore neighborhood, agents in a perception radius. . .).
2. Methods to query the complete list of *local* and *distant* agents on the current process within a `Model`.
3. A method to query the process on which each *distant* `Agent` is executed (for *local* agents, the method returns the current process).

Even if implementation choices are up to the developer, the behavior of each method of the interface for each mode should comply with the statements below:

GhostMode. `read()` and `acquire()` immediately return the local agent state, for both *local* and *distant* agents. The `synchronize()` method updates *distant* agents from data fetched from their origin process. In consequence, only *distant* agents are accessed from a *ghost* copy.

GlobalGhostMode. The `read()` method always returns a *ghost* copy of the agent, even if it is *local.* The `acquire()` method returns the local state of the agent, to allow an agent to update itself, but should not be called on other agents. The `synchronize()` method updates *distant* agents as in *GhostMode*, and updates the *ghost* copy of each *local* agent with its current state.

HardSyncMode. Fetching up-to-date data from distant processes at each *read* or *write* operation, and performing concurrent data modifications within the time step, including on *distant* agents, is allowed. `read()` and `acquire()` methods might hence fetch data from other processes. The `acquire()` method must ensure an exclusive access upon return, while several processes are allowed to simultaneously `read()` an agent. The release methods are used to manage locks according to the well-known Readers-Writers problem. When a *distant* agent is acquired, its complete state is imported on the current process which can perform any modification on it since the access is exclusive. The `releaseAcquire()` method then sends back the new agent state, in order to update the agent on its origin process. Such design allows users to implicitly perform any write operation on *distant* agents, without specifying any specific action or message exchange protocol. Notice that this implies that `read()` and `acquire()` methods might block even when performed on *local* agents, waiting for distant processes to release them. The `synchronize()` method is not required to perform any data update, but should act as a synchronization barrier so that it returns only when all processes have reached the `synchronize()` call.

PushGhostMode. Data access is managed as in *GhostMode* but modifications on *distant* agents are not overridden at the end of each time step but sent back to the origin process by the `synchronize()` method. A user specified conflict resolution mechanism can be implemented to handle updates received from several processes (including the process that owns the agent).

PushGlobalGhostMode. Read access and data updates are performed using a *ghost* copy, as in *GlobalGhostMode.* The acquired *local* agents can however be modified, and pushed back as in *PushGhostMode.*

It is worth noting that the final user, i.e. the MAS modeler, does not need to know anything about the implementation details once a synchronization mode has been successfully implemented within a simulation platform. From the user point of view, the only requirement is to call the functions of the proposed interface. For instance, in the *HardSyncMode*, he just has to call `acquire()` and `releaseAcquire()` to modify any agent as if the agent behavior was performed in a sequential shared memory environment. He does not even need to know whether the modified agent is *local* or *distant.* All calls and concurrency management are automatically and transparently handled in the background by the simulation platform.

3.4 Properties

Several properties can be deduced from the specification of modes, notably in terms of allowed interactions and reproducibility. Such analysis might help modelers to identify which mode can be used depending on their model requirements.

While *GhostMode* and *GlobalGhostMode* prevent write operations on *distant* agents, only the *HardSyncMode* can handle read and write operations at time T on *distant* agents. Performing *write* operations only on *local* agents, as allowed

Table 2: Reproducibility Levels

Level	Description
0	No reproducibility requirement.
1	Results are statistically reproducible.
2	Results are reproducible considering a fixed partitioning.
3	Results do not depend on agents execution order.
4	Results do not depend on partitioning.

by the *GhostMode*, can however be relevant in some contexts, for example when the model is guaranteed to be distributed so that all agents at the same location are assigned to the same process, as in RepastHPC. In that case, predators in the Prey-Predator model could safely eat preys in their current location. However, such modes cannot handle the case when agents are required to perform modifications within their Moore or Von Neumann neighborhood in a grid environment for example, or when agents evolve in a continuous or even non-spatial environment. In the Prey-Predator context, this would require predators to eat *distant* preys executed on other processes, what is not allowed by *GhostMode* or *GlobalGhostMode*. Moreover, since several predators might try to simultaneously eat a same prey within the time step, the *HardSyncMode* is the only mode that can support such interactions. However, if the model rules state that predators *try* to eat preys and wait until the next time step to know if their attack was successful, as in the Influence-Reaction scheme, the model can be simulated using *PushGhostMode* or *PushGlobalGhostMode*. Also notice that the Flocking model, that can be simulated with *GhostMode* or *GlobalGhostMode*, might also be simulated using the *HardSyncMode*: in this case, the position of the *distant* agents will be read directly from the distant processes, not from a local *ghost* copy, what might be a behavior much closer to a sequential execution.

On the other hand, the reproducibility of simulations is directly influenced by the temporal aspect of interactions. This criteria notably motivated the D-MASON authors to implement a *Global Ghost* synchronization. Indeed, reproducibility levels can be defined as specified in table 2. It can be shown that each level is a necessary condition to the next level. Note that the "fixed partitioning" condition requires that a fixed count of processes is also used.

Agent interactions, and thus model results, clearly depend on agents execution order when *reads* and *writes* are performed at time T, since agents perceive modifications in the current time step only for agents executed before them. In consequence, modes allowing such interactions cannot go beyond the level 2 reproducibility. On the other hand, since reads in *ghost* data, i.e. at time $T-1$, do not depend on the execution order of agents, it is possible to reach levels 3 and 4 using *Global Ghost* based modes. Notice that the *ghost* usage is however not sufficient to guarantee such reproducibility levels, since random number generation at the scale of each agent must also not depend on agents execution order or partitioning. Providing this feature is actually not trivial, and even impossible in some contexts. A solution, not discussed in this work, as yet been implemented for grid based models in the FPMAS platform that we use for the experiments.

4 Experiments

The objective of the presented experiments is to compare the synchronization mode performance, using a reference model so as to give to the model developers indications on the impacts of the chosen synchronization mode.

4.1 Experimental Settings

Experiments are based on the well known SIR epidemiological model. The advantage of this model is that it can run properly with all the synchronization modes introduced in this paper. It consists in a grid where agents randomly move and perceive other agents in their Moore neighborhood at each time step. At each time step, *Infected* agents can infect each of their *Susceptible* neighbor with a probability β, can *Recover* with a probability γ, or die with a probability μ. Model sources and more detailed information can be accessed online [Bre22b]. From such model specification, two versions of the model can be implemented:

- A read-only version, where each agent performs a Bernoulli experiment of parameter β with each of its *Infected* neighbor: if the result of at least one is positive, the agent gets *Infected*.
- A write version, where *Infected* agents directly infect their *Susceptible* neighbors with a probability β.

The read-only version can run properly with *GhostMode*, *GlobalGhostMode* and *HardSyncMode*, even if this might implicitly result in different interactions. For example, in *GhostMode*, *local* agents already *Infected* within the time step will be perceived as *Infected*, while *distant* agents *Infected* in the current time step will still be perceived *Susceptible* until the next time step. In *HardSyncMode*, *distant* agents *Infected* in the current time step will immediately be perceived as *Infected*, while in *GlobalGhostMode* all infections are only perceived at the next time step. The *write* version could run properly with *HardSyncMode*, *PushGhostMode* and *PushGlobalGhostMode*.

For the experiments we use the C++ FPMAS [Bre22a] platform, that allows to transparently distribute MAS model simulations. The SIR model is distributed using a grid based load balancing. Currently only the *GhostMode*, *GlobalGhostMode* and *HardSyncMode* are implemented but *PushGhostMode* and *PushGlobalGhostMode* could easily be implemented. While *GhostMode* and *GlobalGhostMode* are relatively trivial to implement, the *HardSyncMode* introduces a complex architecture to provide features that have not been encountered in existing platforms. In consequence, some of the *HardSyncMode* implementation details within FPMAS are provided as a contribution. The solution is based on a distributed client/server architecture. Each process is attached to a client and a server instance. Each server handles requests to its *local* agents, while the client sends requests to *distant* agents to other processes. When `read()` or `acquire()` is called on a *local* agent, the local process possibly waits for other processes to release it, while processing requests of other agents in order to ensure progress. In order to prevent deadlocks, requests are performed using non-blocking communications: while waiting for a response, the local process keeps handling incoming requests. This allows a deadlock free single thread *HardSyncMode* implementation, but other solutions based on multi-threading could be designed. The `synchronize()` method then consists in handling incoming requests until all processes have initiated the `synchronize()` method,

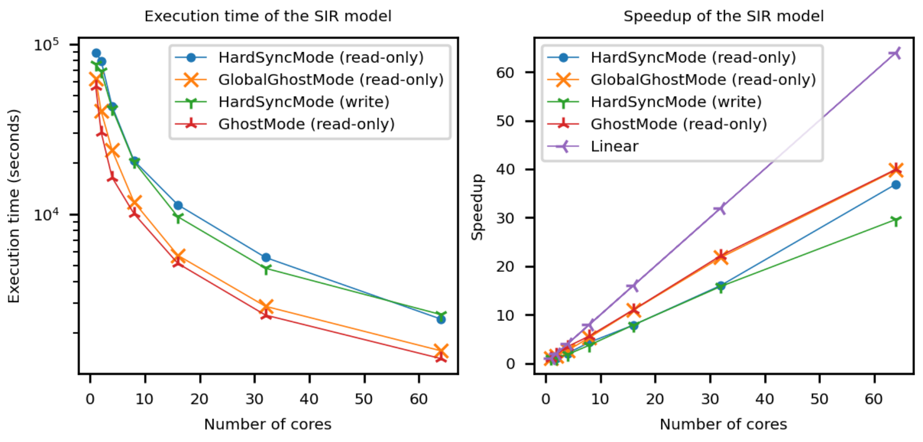

Figure 1: Execution times (log scale) and speedups for a SIR model instance with $1,000,000$ agents on a 1500×1500 grid, with read-only and write versions. The speedup is computed for each mode as $\frac{T_n}{T_1}$ where T_n represents the execution time with n cores.

i.e. all agents have been executed.

4.2 Results

We note that the objective here is only to provide a relative performance comparison of modes, so we do not compare the FPMAS *HardSyncMode* with the RepastHPC *GhostMode* equivalent synchronization for example.

Execution times for a SIR model instance, depending on the number of cores, are presented on figure 1. Experiments were run on the local computing center, on Intel Xeon 6126 processors running at 2.60GHz. Each measure is the average of 10 runs of 1000 time steps with different seeds: execution time variations are negligible. First we can observe that with the *GlobalGhostMode* the execution time of the model with $1,000,000$ agents on a 1500×1500 grid drops from 17 hours in sequential to 26 minutes using 64 processes, what illustrates the point of using HPC resources to execute MAS simulations. The cost of the *GlobalGhostMode* can be explained by the required copies of all agents at the end of each time step. In comparison, the *GhostMode* only needs to perform copies on *distant* agents, with the same amount of communication. The cost of the *HardSyncMode* comes from the point to point communications that are required for each *read* or *write*. As observed on the speedup curves, such communications scales less than other modes, especially considering the fact that more cores means more *distant* agents and so more communications since the number of agent is constant. The *GhostMode* and *GlobalGhostMode* scale well because they only require well balanced collective communications.

The results of the write version of the SIR model obtained with *HardSyncMode* on 64 processes are presented on figure 2 and shows the consistency of a simulation performing *distant writes* with *HardSyncMode*. Even if this mode is the most costly, we ran the simulation with $7,000,000$ agents and a 3000×3000 grid on 64 cores for 500 time steps in 3 hours, which show that this mode can practically be used.

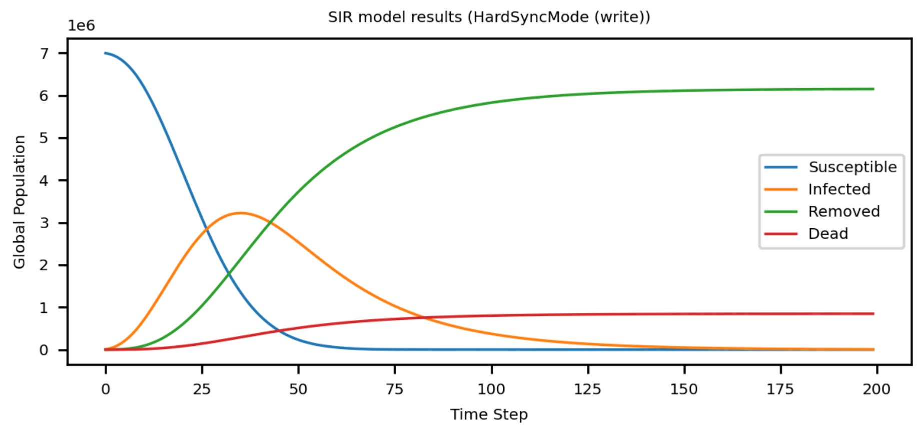

Figure 2: Example results with $7,000,000$ agents and a 3000×3000 grid on 64 processes (cropped to 200 time steps).

Note that the results are not limited to the chosen platform, and the relative comparison of modes should hold independently of their implementation. FP-MAS indeed allows to build meaningful benchmarks, since the implementation of each mode is well optimized and the synchronization mode can be changed at the model level, without altering the read and write SIR model implementations [Bre22b], preventing implementation bias.

5 Conclusion

To address the data synchronization issue in distributed MAS simulations, several synchronization modes with different properties have been presented. The generic definition of the problem allows to define a common interface to implement models using generic operations, without considering distributed memory issues. This interface also allows to inject different synchronization modes in the same model, without changing its implementation, what is particularly useful to produce robust performance and results benchmarks. In this work, we notably focused on a theoretical analysis of proposed modes, and on their performances. It appears that even if the *GhostMode* does not provide the best properties in terms of reproducibility and does not allow remote writes, it is very efficient in terms of performance. On the other hand, the *HardSyncMode* is slower, but is currently the only mode that provides concurrent and remote *reads* and *writes* within the current time step. Finally, the *GlobalGhostMode* is relatively costly and does not allow remote writes, but it is the only mode that can guarantee reproducible results independently of the model distribution. Such considerations show how the synchronization mode used to execute a MAS simulation on HPC resources is relevant, and should depend on models and user requirements, justifying the need for platforms that propose several well defined modes.

In our future works we plan to realize a detailed experimental analysis about the model results and reproducibility, in order to study the synchronization mode impacts on model results. Distinct or specific model needs might also motivate the definition of new modes, formally specified with the generic interface.

Acknowledgments

Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté. FPMAS [Bre22a] and model sources [Bre22b] are accessible online thanks to permanent identifiers provided by Software Heritage.

References

[BGLS17] Francisco Borges, Albert Gutierrez-Milla, Emilio Luque, and Remo Suppi. Care HPS: A high performance simulation tool for parallel and distributed agent-based modeling. *Future Generation Computer Systems*, 68:59–73, March 2017.

[BHLP21] Paul Breugnot, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. A Synchronized and Dynamic Distributed Graph structure to allow the native distribution of Multi-Agent System simulations. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 54–61, March 2021.

[Bre22a] Paul Breugnot. FPMAS Platform v1.5.1. FEMTO-ST, April 2022.

[Bre22b] Paul Breugnot. FPMAS Virus Model v1.0. FEMTO-ST, April 2022.

[BT19] Jim Blythe and Alexey Tregubov. FARM: Architecture for Distributed Agent-Based Social Simulations. In Donghui Lin, Toru Ishida, Franco Zambonelli, and Itsuki Noda, editors, *Massively Multi-Agent Systems II*, Lecture Notes in Computer Science, pages 96–107, Cham, 2019. Springer International Publishing.

[CN12] Nicholson Collier and Michael North. Parallel agent-based simulation with Repast for High Performance Computing. *SIMULATION*, November 2012.

[CSS16] Gennaro Cordasco, Carmine Spagnuolo, and Vittorio Scarano. Toward the New Version of D-MASON: Efficiency, Effectiveness and Correctness in Parallel and Distributed Agent-Based Simulations. In *2016 Int. Parallel and Distributed Processing Symp. Workshops*, pages 1803–1812, May 2016.

[CWG$^+$12] L. S. Chin, D. J. Worth, C. Greenough, S. Coakley, M. Holcombe, and M. Kiran. FLAME : An approach to the parallelisation of agent-based applications. *Rutherford Appleton Laboratory Technical Reports*, (RAL-TR-2012-013), 2012.

[DvD08] Christophe Deissenberg, Sander van der Hoog, and Herbert Dawid. EURACE: A massively parallel agent-based model of the European economy. *Applied Mathematics and Computation*, 204(2):541–552, October 2008.

[MCO$^+$18] Charles M. Macal, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, and John T. Murphy. CHISIM: an agent-based simulation model of social interactions in a large urban area. In *2018 Winter Simulation Conference (WSC)*, pages 810–820, December 2018.

[Mic07] Fabien Michel. The IRM4S model: The influence/reaction principle for multiagent based simulation. In *Conf. on Autonomous Agents and Multiagent Systems*, AAMAS '07, pages 1–3, Honolulu, Hawaii, May 2007. ACM.

[MS12] Philippe Mathieu and Yann Secq. Environment updating and agent scheduling policies in agent-based simulators. In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence*, volume 1, pages 170–175. SciTePress, 2012.

[PRH$^+$03] Konstantin Popov, Mahmoud Rafea, Fredrik Holmgren, Per Brand, Vladimir Vlassov, and Seif Haridi. Parallel agent-based simulation on a cluster of workstations. *Parallel Processing Letters*, 13(04):629–641, December 2003.

[Rub14] Xavier Rubio-Campillo. Pandora: A Versatile Agent-Based Modelling Platform for Social Simulation. In *Proceedings of SIMUL*, pages 29–34, January 2014.

[SDHP10] David Scerri, Alexis Drogoul, Sarah Hickmott, and Lin Padgham. An Architecture for Modular Distributed Simulation with Agent-Based Models. In *Int. Conf. on Autonomous Agents and Multiagent Systems*, volume 1, pages 541–548, Toronto, Canada, 2010.

[STL13] Vinoth Suryanarayanan, Georgios Theodoropoulos, and Michael Lees. PDES-MAS: Distributed Simulation of Multi-agent Systems. *Procedia Computer Science*, 18:671–681, January 2013.