RAMSES: Reversibility-Based Agent Modeling and Simulation Environment with Speculation-Support

Davide Cingolani, Alessandro Pellegrini(✉), and Francesco Quaglia

DIAG, Sapienza University of Rome, Rome, Italy
cingodvd@gmail.com, {pellegrini,quaglia}@dis.uniroma1.it

Abstract. This paper presents RAMSES, a framework for easily specifying agent-based discrete event models entailing both environment and agent entities. RAMSES offers parallel execution capabilities based on speculative event processing and an innovative software reversibility technique that copes with state restore in case the run slides along a non-consistent speculative path. Reversibility in RAMSES relies on transparent static software instrumentation, thus allowing the model developer to concentrate on the actual forward-execution logic of the simulation events occurring in the system. An experimental assessment of RAMSES is also presented, which is aimed at determining its run-time effectiveness and its potential for simplifying the development of agent-based models when compared to other (general purpose) speculative frameworks for parallel discrete event simulation.

1 Introduction

Agent-based modeling exhibits an intrinsic expressive power, making it a proven solution to study complex real-world scenarios, such as disaster rescue [1], computational sociology [2], biomedical applications [3], and economic analysis [4]. At the same time, discrete event models are mainstream formalisms for describing agent-based models, just due to the fact that agents' interactions with other entities can be abstracted as occurring at specific time instants[1].

Also, discrete event simulation techniques represent a core support for solving agent-based models relying on the discrete event paradigm. This is an important aspects, given the existence of a plethora of techniques, globally referred as Parallel Discrete Event Simulation (PDES) [5], which provides protocols and mechanisms for running complex discrete event simulation models in parallel, hence allowing for speedup in the model execution and tractability of highly complex and/or large/huge models.

However, except for a few specific cases, most of the traditional PDES platforms are general-purpose. Nevertheless, when considering agent-based simulation, the peculiarities of this kind of simulation models can be exploited in

[1] Interactions having a specific duration can be anyhow mapped to a couple of *begin* and *end* discrete events.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 466–478, 2015.
DOI: 10.1007/978-3-319-27308-2_38

order to tailor the PDES environment API, its internal structure and run-time behavior to the actual needs of agent-based scenarios. In particular, two different types of simulation objects/entities can be considered as core building blocks: the *environment*, and the actual *agents* (both either physical or logical).

In this paper we present RAMSES—Reversibility-based Agent Modeling and Simulation Environment with Speculation-support[2]— which has been conceived by starting from the conjecture that the environment can represent the *dominant* objects' category in a wide set of agent-based simulation models. In fact, the environment can experience changes which are either independent of or dependent on the actual agents behavior, and at the same time agents interact among each other within (a portion of) the environment. We exploit this conjecture of *dominance by the environment* to set up a PDES-based execution framework in which the actual evolution of the system is driven only by discrete events which affect portions of the environment—in fact, agent/agent and agent/environment interactions can be easily mapped to events which take place within the environment only. This will be reflected, particularly, in the API offered by RAMSES to the simulation model developer.

Concerning run-time efficiency, RAMSES specifically targets parallel execution on multi/many-core environments, where the simulation's execution is partitioned among different worker-threads that share the overall simulation workload. Also, the approach to synchronize the activities of the different worker-threads to ensure a causally-consistent evolution of the simulation model (global) state, allows for *speculative event processing*, a technique that has been shown to provide high scalability in disparate execution environments (see, e.g., [6]). As for this aspect, RAMSES incorporates an innovative support for correct state restore in case of miss-speculation (namely, a posteriori detection of out-order execution actually affecting causal consistency), which is based on the software reversibility approach recently presented in [7] and has been shown to have the potential to provide overhead reduction, with respect to classical (e.g. checkpoint-based [8]) state recovery. This is especially true for execution patterns of individual events entailing a reduced amount of simulation state updates, independently of the complexity of the actual logic of the events. This might be the case of agent models, since agents might inspect (read) large chunks of data from the environment, while updating the environment state with a limited number of operations (at least in the likely case).

To assess the viability of our proposal, we rely on a set of experiments carried out by using a distributed multi-robot exploration and mapping simulation model developed on the basis of the results in [9]. We compare the performance of RAMSES with an implementation of the same simulation model on top of ROOT-Sim [10], namely a highly-optimized general-purpose simulation framework offering speculative execution capabilities for PDES models. We also consider software development aspects in the comparison (e.g. required number of

[2] The source code of RAMSES can be found at https://github.com/HPDCS/RAMSES

code lines for the same model in the different—special vs general purpose—frameworks).

The remainder of this paper is structured as follows. In Sect. 2, related work is discussed. The internal structure of RAMSES and the exposed API are described in Sect. 3. The experimental study is presented in Sect. 4.

2 Related Work

We can find a large number of frameworks to support agent-based simulation in the literature. The MASON framework [11] pays special attention to the performance of simulation execution, addressing computing-intensive models (i.e., large 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, relying on the innovative generation of update undo code blocks for the reconciliation of causality errors.

Pandora [13] 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). On the other hand, RAMSES provides the simulation model developer with an API that is specifically tailored for implementing simulation models in ANSI-C, which binds the interactions to the environment. This allows for a simplified implementation of simulation models, giving transparently access to highly optimized synchronization facilities to support efficient computations on multi/many-core machines.

AnyLogic [14] 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 C technology and rely on innovative synchronization protocols to carry on the simulation work. Additionally, we provide an API specifically targeting agent-based models, allowing for an easy implementation of simulation models.

FLAME [15] 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 [16] 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.

RAMSES is naturally related to the literature dealing with parallel speculative processing. This paradigm is recognized as a means to achieve scalability thanks to the (partial) removal of the cost for coordinating concurrent processes and/or threads from the critical path of task processing (see, e.g., [17]).

In the PDES context, the speculative paradigm is incarnated by the well-known Time Warp synchronization protocol [18], which has been recently shown to provide scalability up to thousands or millions of CPU-cores [6]. Among the PDES platforms that have been developed by relying on Time Warp synchronization, a close one to RAMSES is ROOT-Sim [10]. This is a general-purpose speculative simulation framework for discrete-event simulation models that has been recently enhanced with the software reversibility support we exploit in RAMSES. Although ROOT-Sim has been proven [19,20] to be able to efficiently carry out the execution of agent-based models, it does not account for the requirements of developing this kind of models. Conversely, in RAMSES, we offer a specific API to let the model writer implement agent-based models more easily. Additionally, the internal runtime execution support which we propose in this paper is based on an innovative synchronization protocol different from the more traditional Time Warp-based one of ROOT-Sim.

3 RAMSES

3.1 Reference Programming Model

We target a programming model in which two different types of entities compose the overall structure of the simulated phenomenon. On the one side we have the environment, which could be of any size and shape, yet we expect it to be divided into regions. Each region has the following characteristics:

- A region has a state, which describes all the aspects of the environment. We do not place any limit on the number and kind of aspects that a region might have, thus giving the model programmer the highest degree of freedom in the definition of the environment.
- A region might host one or multiple agents at a given time instant.
- The internal state of the region might be modified either by external circumstances (e.g., an earthquake might change the shape of the terrain), or by the interaction with one or more agents (e.g., one agent might drop or collect objects into a region).
- Regions can be logically connected to each other depending on the actual model's logic, thus the environment is customizable in size and shape.

On the other side we have agents, which adhere to the following behavior:

- Agents have a state, which describes all the aspects of their current evolution within the environment. This state can describe either physical characteristics (e.g., the conditions of some parts of a mechanical agent) or logical/cognitive characteristics (e.g., the knowledge of the environment that an exploring agent has gathered so far).
- Agents are always located in a region. We note that this does not pose a limitation to the programming freedom of the model developer, as logical dummy regions can be used to host agents, in case the model requires agents to be outside of the environment, at a given point in the simulation.

– Agents can move freely in the environment, yet moving only across regions which are adjacent. Again, this does not pose a limitation to the programming freedom, as any region can be connected to any other region, depending on the model's logic.
– Agents can interact with each other. Nevertheless, this interaction can only take place when two agents are located within the same region.
– Agents can interact with the environment. In particular, they can inspect the environment to gather knowledge, or they can modify it.

We emphasize that (as already hinted) all the interactions take place within regions. Therefore, changes in the state of any entity (be it a region or an agent) which are produced by the events' execution take place according to the (simulated) time advancement of regions. We call this property *dominance by the environment*, which will be exploited by the architectural organization presented in Sect. 3.4. In particular, since our simulation framework executes according to a PDES scheme, we map only regions to the traditional notion of simulation objects. In this way, we are able to significantly reduce the amount of entities which are managed by the multi-threaded simulation infrastructure, and the associated management/synchronization costs.

3.2 Exposed API

The API exposed by RAMSES includes functions of various nature, which can be grouped into two main categories: functions to model the initial state of the simulation (in terms of description of the environment and agents) and to carry on the evolution of the system, and library functions to manipulate the topology of the environment and retrieve the simulation state of regions and agents. For the sake of space constraints, we refer to the online RAMSES documentation for a comprehensive and technical description of the whole set of API functions and supported topologies. We discuss here the basic functions, to let the reader understand the principles driving the implementation of models on top of RAMSES.

To setup and start the simulation, the model writer issues a call (possibly from `main()`) to the `Setup()` API, which allows to specify both the number of regions and agents, along with two *initialization callbacks*, say function pointers, one for the regions and one for the agents. Both pointers refer functions accepting one integer as input (which is the id number of the agent or the region, assigned by the engine) and must return a pointer to the (allocated) simulation state, which will be then managed by the simulation framework. During the initialization of an agent, a call to `void InitialPosition(unsigned int region)` must be issued, so as to specify in which region the currently being initialized agent is placed. Failing to do so results in a runtime error. After the call to `Setup()`, the simulation application can call `void StartSimulation(int n_cores)`, which creates n parallel worker-threads and carries on the simulation execution.

Concerning interactions among agents and the environment, four specific API functions allow to inject in the system new events. They are:

- `void Move(uint32_t agent, uint32_t dest, simtime_t time)`: by issuing a call to this function, the model tells the simulation framework that at a certain time (denoted by `time`) the agent `agent` is moving from the current region which is hosting it to the region identified by `dest`.
- `void AgentInteraction(uint32_t agent_a, uint32_t agent_b, simtime_t time, interaction_f agent_int, void *args, size_t size)`: this function tells the simulation framework that at time `time` two agents, `agent_a` and `agent_b` want to interact. The actual interaction is modeled by the function pointed by `agent_int`, which receives as argument the buffer pointed by `args`. If at time `time` the two agents `agent_a` and `agent_b` are not in the same region, a runtime error is issued.
- `void EnvironmentInteraction(uint32_t agent, uint32_t region, simtime_t time, interaction_f environment_int, void *args, size_t size)`: in case one agent wants to interact with the surrounding environment, namely with a region, this function tells the simulation engine that at time `time` the agent `agent` is expected to be found in region `region`. If this is not the case, a runtime error is issued. On the other hand, the model code receives a call to the function pointed by `environment_int`, having `args` as the parameter.
- `void EnvironmentUpdate(uint32_t region, simtime_t time, update_f environment_update, void *args, size_t size)`: to model updates to the environment which are not related to the interaction with agents, this API function can be used to notify that at time `time` an update to `region` should be carried out by the function pointed by `environment_update`.

Concerning the topology, RAMSES offers several API functions to organize the regions in common topologies. As an example, in case of random movements, by calling `int FindRegion(int type)` the model can receive the id of a neighbor region according to a given topology organization described by `type`.

3.3 Tracking Memory Updates for Reversibility

We rely on software instrumentation to transparently modify the application-level code, in order to let RAMSES engine track at runtime what are the *effects* of the forward execution of events on the simulation model's state. This information is used to build a packed version of negative instructions which only undo the effects of the forward execution, allowing for a reverse execution of events (in terms of state updates) which is independent of the actual forward event granularity (CPU requirement). The whole approach is based on the coexistence of dual executable modes (inspired to [19]), to quickly switch between two operative modes: one tracking memory updates, one which does not.

To statically instrument the application-level code, we rely on the open-source Hijacker [21] tool. By relying on Hijacker, we can specify (via a set of xml-based rules) what are the instrumentation steps to be undertaken before the final linking stage of the application is executed[3]. More specifically, we instruct Hijacker to create multiple copies of the same executable, but differently instrumented. This technique, known as *multi-coding*, creates different versions of the application which nevertheless share the same data sections within the virtual address space, but can be accessed using ad-hoc altered function names. One copy of the software is left untouched (namely, no instrumentation is applied to it). Therefore, this first version can be regarded as the original code, which therefore does not provide any possibility to undo the effects on memory (in terms of updates) by the execution of an event.

The second version, on the other hand, is managed by Hijacker so that the whole simulation model's code is scanned to find assembly instructions which have a memory address as the destination operand[4]. Before each memory-write instruction, Hijacker places a call to a specific internal trampoline, along with some instructions which generate an *invocation context* for it. This trampoline function computes the ultimate target memory address which will be accessed in write mode, and the size of the writing. The couple $\langle address, size \rangle$ is then passed as input to the internal `reverse_generator(void *address, size_t size)` module of RAMSES, which generates the reverse code instructions, just before any memory-update operation is performed.

These *negative* instructions are simply built by accessing memory at `address` and by reading `size` bytes. Since the invocation of `reverse_generator` happens right before the execution of the original memory-write instruction, this allows the module to retrieve the "old" value of the memory location belonging to the simulation model's state. Therefore, it is used to build a *negative* data movement instruction whose destination is `address`. Generally, the generation of negative instructions is not a costly operation as all the opcodes are known beforehand, allowing to use pre-compiled tables of instructions, where only the relevant parameters should be packed within, namely the old memory value and the destination address. To keep the negative instructions packed, each worker-thread operating within the RAMSES simulation engine relies on a (heap-allocated) *reverse window* structure. This structure allows to immediately determine the memory position at which a negative instruction must be stored. In particular, after having allocated at startup the reverse window, instructions are generated in a top-heading stack. This solution allows for a fast annihilation of the effects of an event, as simply jumping to the last-generated instruction (namely, the one on the top of the stack) is enough to execute the negative instructions in reverse order. This is done by issuing a `call` to the address of the top-standing instruction, thus undoing the effects until a final `ret` instruction placed at the

[3] Hijacker works on *relocatable object files* (specifically on the Executable and Linkable Format—ELF), therefore it must be regarded as an additional compilation step of the executable building procedure.

[4] The most significant instructions for the x86 architecture (which represents our target) are `mov`, `movs`, and `cmov` instructions, and are handled internally by Hijacker in different ways. The same is true for vectorized memory access instructions such as `movdqa`.

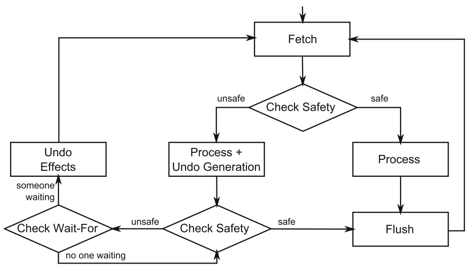

Fig. 1. Main loop flow chart

buffer end is found. Clearly, a unique reverse instruction for any different memory location that is touched by the simulation event in write mode is inserted in the reverse window (namely, the reverse instruction associated with the first update occurring on that location while processing a single event).

3.4 Runtime Execution Support

We consider a scenario where all the scheduled events (including the simulation startup events), destined to whichever simulation object, are kept within a unique pool. For efficiency reasons, we consider a classical calendar queue [22], which provides average $O(1)$ performance. The ordering of the elements into the calendar queue is based on events' timestamp. Each event also keeps information regarding which is the target simulation object (namely the region) and the actual event's type and payload. Nevertheless, the event type is not defined by the model developer. Rather, with respect to the API discussed in Sect. 3.2, the type is internally used by the engine to determine what is the type of callback to be invoked. All the events kept in the calendar queue are *schedule-committed* (hence non-retractable). In fact, in our approach the events that are scheduled during the processing of an event according to the reversibility scheme are only flushed upon the detection of the event *safety* (i.e. causal consistency).

The calendar queue data structure is coupled with an array of N entries that we name `processing[]`. The i-th entry is used to keep data related to the status of the i-th worker-thread, noted WT_i. This array is initialized at simulation startup with all the entries keeping the special value ∞. Each worker-thread WT_i follows through the algorithmic actions depicted in Fig. 1. It performs the FETCH operation presented in Algorithm 1, which (in case the *last_event* input parameter, indicating whether an event still to be processed is already bound to this thread, is $NULL$) atomically extracts the event e with minimum timestamp that is currently registered into the calendar queue, and records the extracted timestamp value into the entry `processing[i]` associated with WT_i. Atomicity ensures that an event is taken by only one thread.

Due to the multi-threaded nature and the speculative flavor of our simulation environment, no two different worker-threads can execute at the same time multiple events which entail reading/updating the same memory regions.

Algorithm 1. Fetch procedure - worker-thread WT_i

```
1: procedure FETCH($last\_event\ e$) RETURNS: event
2:     if $e = NULL$ then
3:         SPINLOCK(global_lock) //this branch is atomic via a globally shared lock
4:         $e \leftarrow$ GETMINIMUMTIMESTAMPEVENTFROMCALENDARQUEUE( )
5:         processing[i] $\leftarrow T(e)$
6:         SPINUNLOCK(global_lock)
7:     end if
8:     if $\neg$TRYLOCK(region_lock[$e.destination$]) then
9:         repeat
10:            $reupdateMin \leftarrow$ false
11:            $minWait \leftarrow$ wait_time[$e.destination$]
12:            if $T(e) < minWait$ then
13:                if $\neg$ CAS(wait_time[$e.destination$], $minWait$, $T(e)$) then
14:                    $reupdateMin \leftarrow$ true
15:                end if
16:            end if
17:        until $reupdateMin$
18:        while TRUE do
19:            SPINLOCK(region_lock[$e.destination$])
20:            if $T(e) \leq$ wait_time[$e.destination$] then break
21:            end if
22:            SPINUNLOCK(region_lock[$e.destination$])
23:        end while
24:    end if
25:    return $e$
26: end procedure
```

Therefore, to enforce data separation, we rely on an array of spinlocks, which we call `region_lock[]`. Whenever WT_i fetches an event e from the calendar queue, it tries to acquire the lock for the given recipient simulation object (see Algorithm 1). In case the lock cannot be taken, it means that another worker-thread is currently executing operations on the region's state. In this case, the worker-thread spins on the lock, until the other worker-thread completes its operations. We note that, due to the region dominance property defined in Sect. 3.1, this sanity check ensures as well that all worker-threads access the agents' states in data separation, as no agent can be (at the same time) in two different regions.

When an event is processed, new events possibly generated by the processing actions are temporarily buffered (hence not yet flushed to the calendar queue). However in case they eventually become schedule committed, then the FLUSH procedure is called, which atomically inserts them into the calendar queue.

To cope with consistency, and to determine event processing commitment and event schedule commitment, we exploit the values kept by the `processing[]` array. The condition that tells whether a worker-thread WT_i can safely commit the event it is handling is $\forall j \neq i : \texttt{processing}[\texttt{i}] < \texttt{processing}[\texttt{j}]$. This condition tells that the (possibly speculatively) executed event is associated with the current lower bound timestamp across all the events in the system[5]. Hence the timestamp of this event represents the commit horizon, and the event can be

[5] The case of simultaneous events, where $T(e)$ may be equal to $T(e')$, can be addressed using a variant of Lamport's bakery algorithm [23] either including causality information or simply thread identifiers.

safely executed or (in case of already carried out speculative execution) safely committed.

Concerning execution liveness, if the `region_lock` is taken by any worker-thread speculatively processing an event e associated with $T(e)$, and during its execution a new event e' associated with $T(e') < T(e)$ is flushed (e.g., by a worker-thread executing in non-speculative mode) into the calendar queue, we might incur in livelock. Therefore, an additional array `wait_time[]`, with one entry for each managed region, is used to notify worker-threads running in speculative mode that an event with higher priority is waiting to be processed by another worker-thread (see Algorithm 1 for the logic used to post the event timestamp value within this array entries). As depicted in Fig. 1, if a worker-thread has executed speculatively an event which is (not yet) safe, it checks whether any other worker-thread has registered within the `wait_time[]` array a timestamp which is less than that of the event currently being processed. In the positive case, the effects of the event's execution are undone (by simply jumping to the generated reverse window) and the region lock is released. In this case, the event stays bound to the worker-thread (for re-processing), and is passed in input to FETCH, which will skip extracting another event from the calendar queue. On the other hand, in case of commitment of the processed event, a $NULL$ value is passed in input (as last-event record) to the FETCH procedure.

4 Experimental Results

To assess the programmability and performance of RAMSES, we have implemented a distributed multi-robot exploration and mapping simulation model, according to the results in [9]. A group of robots is set out into an unknown space to fully explore it, while acquiring data from sensors to map the environment. Whenever a robot has to make a decision about which direction should be taken to carry on the exploration, it is done by relying on the notion of *exploration frontier*. By keeping a representation of the explored world, the robot is able to detect which is the closest unexplored area which it can reach, computes the fastest way to reach it and continues the exploration. The robots explore independently of each other until one coincidentally detects another robot. In this case, they exchange the data acquired during the exploration, thus reducing the exploration time and allowing for more accurate decisions. We have implemented this model on top of both RAMSES and ROOT-Sim[6]. The latter is an open source PDES simulation engine developed using C/POSIX technology, still targeting speculative processing. Differently from RAMSES, it is general purpose, thus offering an API based on a single application entry point, representing the event handler. ROOT-Sim transparently supports all the mechanisms associated with parallelization (e.g., mapping of simulation objects on different kernel instances) and optimistic synchronization (e.g., state recoverability). This allow us to compare the efficiency of our innovative runtime execution support

[6] The ROOT-Sim version is the one used as test-bed in [8].

against an already highly-optimized, but general purpose, simulation framework for PDES, targeting as well multi/many-core architectures.

We simulated an environment composed of 4096 regions, and we varied the number of agent (robot) units moving around between 100 and 1000, which allowed us assessing how the performance of RAMSES scales vs variations of the ratio between the number of regions and the number of agents.

In Fig. 2 we report data for a comparison of the performance achieved via RAMSES and ROOT-Sim with the one achieved via sequential execution of the simulation model (on top of a calendar queue scheduler)[7]. All the experiments have been carried out on an HP ProLiant server, equipped with four 2 GHz AMD Opteron 6128 processors and 64 GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12 MB L3 cache (6 MB per each 4-cores set), and each core has a 512 KB private L2 cache. The architecture entails 8 different NUMA nodes. The operating system is 64-bit Debian 6, with Linux Kernel version 2.6.32.5.

By the results, when the number of robots is small (namely, 100 robots), the speedup offered by RAMSES over the sequential run is low, while ROOT-Sim provides definitely better reduction of the execution time. However, for large numbers of simulated robots (namely, 1000 robots), RAMSES starts becoming competitive with respect to ROOT-Sim, by providing a speedup over the sequential run of about 15. We deduce that this performance trend is directly linked to how the simulation engine manages event concurrency. Particularly, in ROOT-Sim the robots are modeled as purely concurrent entities, which leads to the fact that if multiple robots collide within the same region, the associated events can still be processed concurrently. Instead, in RAMSES, if multiple robots temporarily reside within the same region, then all their events are sequentialized given that they are mapped to region-events at the level of the reversibility-based speculative underlying engine. However, for larger numbers of robots, we get higher likelihood that multiple regions hosting robots can be scheduled concurrently by the RAMSES engine. On the other hand, the ratio between the number of application level code lines to implement the model in RAMSES and in ROOT-Sim is of the order of 0.65, which roughly indicates 35 % reduction of the application code complexity for implementing the same model, achieved thanks to the agent-modeling suited API offered by RAMSES. Overall, RAMSES can exemplify the development of agent-based models (as compared to what allowed by a classical general purpose PDES engine) while still providing run time efficiency especially for more complex models (namely with larger ratio between active and passive objects).

[7] The sequential code version exactly corresponds to the one run on top of ROOT-Sim. However, a port of the version run on RAMSES on the same sequential engine provided quite similar execution times.

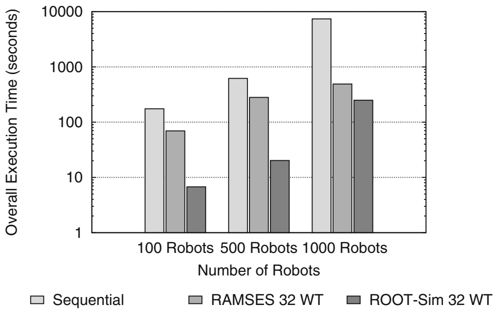

Fig. 2. Experimental results: 4096 regions, varied number of robots.

5 Conclusions

In this paper we have presented RAMSES, a framework for agent-based modeling and simulation, relying on speculative concurrent processing and an innovative synchronization (namely rollback) protocol which exploits reversibility to undo a portion of the speculative simulation which is a-posteriori detected to be inconsistent. The experimental assessment on a case study has shown that RAMSES is performance-effective especially for more complex models and, at the same time, can definitely reduce the complexity of coding agent-models when compared to what can be done on top of general-purpose parallel simulation frameworks.

References

1. Takahashi, T., Tadokoro, S., Ohta, M., Ito, N.: Agent based approach in disaster rescue simulation - from test-bed of multiagent system to practical application. In: Birk, A., Coradeschi, S., Tadokoro, S. (eds.) RoboCup 2001. LNCS (LNAI), vol. 2377, pp. 102–111. Springer, Heidelberg (2002)
2. Macy, M.W., Willer, R.: From factors to actors: computational sociology and agent-based modeling. Ann. Rev. Sociol. **28**(1), 143–166 (2002)
3. Macal, C., North, M.: Tutorial on agent-based modeling and simulation part 2: how to model with agents. In: Proceedings of the 2006 Winter Simulation Conference, WSC, pp. 73–83. Society for Computer Simulation (2006)
4. Page, S.E.: Agent-based models. In: Durlauf, S.N., Blume, L.E. (eds.) The New Palgrave Dictionary of Economics. Palgrave Macmillan (2008)
5. Fujimoto, R.M.: Parallel discrete event simulation. In: Proceedings of the 21st Conference on Winter Simulation, WSC, pp. 19–28. ACM Press (1989)
6. Barnes Jr, P.D., Carothers, C.D., Jefferson, D.R., LaPre, J.M.: Warp speed: executing time warp on 1, 966, 080 cores. In: SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS 2013, Montreal, QC, Canada, 19–22 May 2013, pp. 327–336 (2013)
7. Cingolani, D., Pellegrini, A., Quaglia, F.: Transparently mixing undo logs and software reversibility for state recovery in optimistic pdes. In: Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS. ACM Press, June 2015

8. Pellegrini, A., Vitali, R., Quaglia, F.: Autonomic state management for optimistic simulation platforms. IEEE Trans. Parallel Distrib. Syst. **26**(6), 1560–1569 (2015)
9. Fox, D., Ko, J., Konolige, K., Limketkai, B., Schulz, D., Stewart, B.: Distributed multirobot exploration and mapping. Proc. IEEE **94**(7), 1325–1339 (2006)
10. Pellegrini, A., Quaglia, F.: The ROme OpTimistic simulator: a tutorial. In: Mey, D., et al. (eds.) Euro-Par 2013. LNCS, vol. 8374, pp. 501–512. Springer, Heidelberg (2014)
11. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: Mason: a multiagent simulation environment. Simulation **81**(7), 517–527 (2005)
12. Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., Spagnuolo, C.: A framework for distributing agent-based simulations. In: Alexander, M., et al. (eds.) Euro-Par 2011, Part I. LNCS, vol. 7155, pp. 460–470. Springer, Heidelberg (2012)
13. Wittek, P., 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)
14. Karpov, Y.G.: Anylogic – a new generation professional simulation tool. In: Proceedings of the 6th International Congress on Mathematical Modeling, MATHMOD, September 2004
15. Holcombe, M., Coakley, S., 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)
16. Richmond, P., Romano, D.: Agent based GUP, a real-time 3D simulation and interactive visualisation framework for massive agent based modelling on the gpu. In: Proceedings International Workshop on Supervisualisation (2008)
17. Romano, P., Palmieri, R., Quaglia, F., Carvalho, N., Rodrigues, L.: On speculative replication of transactional systems. J. Comput. Syst. Sci. **80**(1), 257–276 (2014)
18. Jefferson, D.R.: Virtual Time. ACM Trans. Prog. Lang. Syst. **7**(3), 404–425 (1985)
19. Pellegrini, A., Quaglia, F.: Programmability and performance of parallel ECS-based simulation of multi-agent exploration models. In: Lopes, L., et al. (eds.) Euro-Par 2014, Part I. LNCS, vol. 8805, pp. 395–406. Springer, Heidelberg (2014)
20. Pellegrini, A., Quaglia, F.: A study on the parallelization of terrain-covering ant robots simulations. In: Mey, D., et al. (eds.) Euro-Par 2013. LNCS, vol. 8374, pp. 585–594. Springer, Heidelberg (2014)
21. Pellegrini, A.: Hijacker: efficient static software instrumentation with applications in high performance computing (poster paper). In: Proceedings of the 2013 International Conference on High Performance Computing & Simulation, HPCS. IEEE Computer Society, July 2013
22. Brown, R.: Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. Commun. ACM **31**, 1220–1227 (1988)
23. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974)