Programmability and Performance of Parallel ECS-based Simulation of Multi-Agent Exploration Models

Alessandro Pellegrini and Francesco Quaglia

DIAG, Sapienza, University of Rome

Abstract. While the traditional objective of parallel/distributed simulation techniques has been mainly in improving performance and making very large models tractable, more recent research trends targeted complementary aspects, such as the "ease of programming". Along this line, a recent proposal called Event and Cross State (ECS) synchronization, stands as a solution allowing to break the traditional programming rules proper of Parallel Discrete Event Simulation (PDES) systems, where the application code processing a specific event is only allowed to access the state (namely the memory image) of the target simulation object. In fact with ECS, the programmer is allowed to write ANSI-C event-handlers capable of accessing (in either read or write mode) the state of whichever simulation object included in the simulation model. Correct concurrent execution of events, e.g., on top of multi-core machines, is guaranteed by ECS with no intervention by the programmer, who is in practice exposed to a sequential-style programming model where events are processed one at a time, and have the ability to access the current memory image of the whole simulation model, namely the collection of the states of any involved object. This can strongly simplify the development of specific models, e.g., by avoiding the need for passing state information across concurrent objects in the form of events. In this article we investigate on both programmability and performance aspects related to developing/-supporting a multi-agent exploration model on top of the ROOT-Sim PDES platform, which supports ECS.

1 Introduction

Timeliness in the delivery of simulation output is an increasingly relevant issue to cope with, especially in contexts where simulation is exploited as a tool for time-critical decision making. This happens in many fields, ranging from agent based simulation of rescue scenarios [1] to simulation of on-line rescaling/reconfiguration policies of systems deployed in, e.g., Cloud Computing environments [2]. For the case of Discrete Event Simulation (DES) models, performance issues have been traditionally targeted via the Parallel-DES (PDES) paradigm [3], which has been based on the partitioning of the simulation model into distinct simulation objects (also known as Logical Processes - LPs) to be executed

concurrently, so as to allow the exploitation of (large scale) parallel platforms to speedup model execution.

The PDES paradigm laid its foundation on a programming model where the states of the involved simulation objects are disjoint, and where memory access operations (upon event processing) are confined within the state of the simulation object dispatched for processing the event. This approach implicitly requires the application programmers to shift from a sequential programming model where the application is designed and coded to run serially (namely to process one event at a time) and has the possibility to access any valid memory location upon the execution of whichever event. In other words, parallelism is achieved by a-priori, namely at code design/development time, forcing separation of the accesses to slices of the simulation model state, each one representing an individual object.

In order to recover from the scenario where parallelism in the execution is forced to be bound (in the programming model) to memory disjoint accesses to an a-priori partitioned simulation model state, recent design/development trends in PDES systems gave rise to the possibility of sharing state information across the concurrent simulation objects (see, e.g., [4–6]). A very recent result in this area is the Event and Cross State (ECS) synchronization protocol presented in [7], which is based on a proper x86_64/Linux memory management architecture and on user-transparent system-level events' whose processing rules allow to "transactify" the execution of any individual simulation event, even in case it accesses (in either read or write mode) the state of multiple simulation objects. ECS is suited for simulations carried out on machines relying on multi-core technology, which is a mainstream architectural support for parallelism. With ECS we are exposed to a sequential-style programming/execution model based on ANSI-C event-handlers, where any code block of a handler can access any valid memory location within the simulation model state, and events are processed by providing the illusion of a traditional sequential (timestamp ordered) execution, where nothing can happen in memory except for read/write actions related to a single event at a time. On the other hand, simulation events are actually processed concurrently as in traditional PDES systems, thus allowing the possibility to still exploit parallelism in the underlying architecture for performance reasons. ECS has been integrated within the open source ROOT-Sim simulation environment [8], in combination with the native ROOT-Sim support for speculative (optimistic) processing.

In this article we investigate on the effects of ECS on programmability of a multi-agent exploration scenario, as well as on the finally delivered performance while executing the simulation model. We will show how ECS permits coding the model very easily by allowing the combination of kind of active (agents) and passive (regions) simulation objects. The region information can be directly accessed in read/write mode by the agents simulation objects while processing their events (e.g., an access-to-region event) which allows for avoiding (A) the need for marshalling/unmarshalling region information and exchanging it in the form of query/reply events and (B) the need for scheduling region update events

(e.g. command events) destined to be processed by passive region-objects. Further, being the regions distinct (although passive) objects, they still allow for concurrency in their accesses by the different agent objects. Overall, thanks to ECS we can get both a reduction of the complexity of the coding process (e.g. via the avoidance of query/reply events) and a transparent support for efficient parallelization. To the best of our knowledge, this is the first study aimed at assessing the effectiveness of techniques for transparent parallelization of multi-agent models on multi-core machines via the provisioning of support for a truly sequential-style programming model based on ANSI-C. Hence, as a matter of fact, we also implicitly provide indications on how to exploit ECS as the support for coding different kinds of models entailing the need for representing both spatial regions and active entities operating within the regions.

The remainder of this paper is organized as follows. Related work is discussed in Section 2. In Section 3 we provide an overview of ECS. In Section 4 we discuss how to exploit ECS facilities for coding the multi-agent exploration model. Experimental results are provided in Section 5.

2 Related Work

A few studies exist in the area of porting sequential agent-based simulators to parallel platforms (see, e.g., [9]). However, the approach there taken is mostly based on the aforementioned query/reply or command events, which are in some cases handled via wrappers encapsulating the actual application code and intercepting the attempt to access to a remote object state (as in classical RPC architectures). Also, these proposals target object oriented programming languages, while ECS is aimed at transparent parallelizing simulation programs based on ANSI-C. Additionally, as far as we know, the proposal in [9] relies on time-stepped (conservative) synchronization as implemented on top of the facilities offered by the Mozart framework [10], while we target transparent parallelization in the context of asynchronous optimistic discrete event simulation.

Other literature studies (see, e.g., [11–14]) provide techniques and environments for parallel execution of agent-based simulation models. However, their primary focus is not on parallelism transparency (in the form of allowing a sequential style code to be directly processed concurrently). Hence, these proposals stand as orthogonal to what we present in this paper.

3 ECS Overview

The ECS synchronization protocol [7] is based on managing the virtual memory destined for usage to any simulation object according to *stocks*. More in detail, when the object requests new memory buffers for installing and/or updating its state layout, which is supported via the traditional `malloc` service, redirected to a proper memory allocator integrated with ECS, the memory management architecture reserves an interval of page-aligned virtual memory addresses, namely the stock, via the standard `mmap` POSIX API.

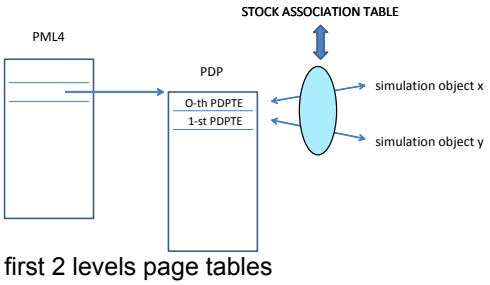

Fig. 1. Example association between stocks of virtual memory and simulation objects.

To understand how ECS manages the stocks, let us consider the actual paging scheme supported by Linux on top of x86_64 architectures, namely the target platform for the ECS protocol. Virtual addresses are used as access keys for a 4-level paging scheme, ultimately managing pages of 4KB in size. The top level page table is called PML4 (or PGD—Page General Directory) and keeps 512 entries. All the other page tables, operating at lower levels, also have 512 entries each. In ECS, the stock of virtual memory pages destined to allocate memory buffers for a given simulation object corresponds to the set of contiguous virtual-pages whose virtual-to-physical memory translation is associated with a single entry of the second-level page table, which is called PDP—Page Directory Pointer (its entries are therefore referred to as PDPTE). Note that a single stock corresponds to 512^2 pages, for a total of 1GB of virtual memory. Hence, a single stock allows managing an object-state requesting up to 1GB of (dynamic) memory. On the other hand, reserving multiple stocks for a same simulation object will lead to managing object states reaching multiple gigabytes in size. In Figure 1 we show an example where a given PDP table has its 0-th entry—and hence the corresponding stock of virtual memory pages—reserved for object x, and its 1-st entry reserved for object y.

The above scheme is complemented by having different worker threads within the simulation process (run on the multi-core machine) associated with multiple *sibling* page tables, which is achieve by augmenting the Linux paging scheme with the facilities offered by a special device file, as designed in [7] (installed via an external module). When a simulation object x is dispatched to be run by worker thread T_i for processing its next event e, then the access rule to the stock associated with x is changed in the corresponding PDP entry of the sibling page tables associated with T_i from `NULL` to a value allowing read/write access by the thread (which is done via an `ioctl` command issued towards the special device file). Instead, the memory stocks associated with the other simulation objects are left unaccessible, in terms of read/write permission. If during the processing of the event e at object x, the event handler run by thread T_i performs an access to a virtual memory region currently in use for the stock associated with object y, ECS captures the access via a lightweight trap-handler (not requiring to pass through the full memory-fault handling chain of Linux) and"transactifies" the

execution of the event by allowing the handler run by thread T_i to access the target stock (opening the access permission on the sibling page tables of T_i) in mutual exclusion. Also, the transaction associated with the event e needs to observe the snapshot of the target object y exactly at the timestamp of e. To achieve this, ECS relies on special system-level rendez-vous events, which support a temporary block of the source/target object along virtual time (at the rendez-vous point). In particular, the source object x remains blocked until the correct snapshot of the target one is available, while the target object y remains blocked in the rendez-vous until the event e is completely processed by object x.

The actual synchronization protocol instantiated in ECS is optimistic, with the meaning that the local simulation clock of the target object y might be beyond the value of the timestamp of the event e at the time when the corresponding transaction (namely rendez-vous) needs to be processed. This is addressed by having rendez-vous events marked with the correct timestamp for their occurrence (namely, the timestamp of e in our example discussion), and by forcing a rollback of the state snapshot of the target object y to a simulation time equal to the timestamp of the rendez-vous in case of out of timestamp order occurrence of the rendez-vous.

As also discussed in [7], the ECS synchronization protocol is deadlock free, thanks to a priority scheme adopted to process events whose timestamp falls in the past of the current logical clock of the target object. In particular, anytime an event in the past needs to be processed by an object which is currently blocked due to a rendez-vous it issued (which still needs to be finalized since the state image of the target object, hit by the rendez-vous, is still unavailable), the block phase is squashed, thus avoiding indefinite block scenarios. On the other hand, in case the event to be processed is scheduled at the same time of the blocking rendez-vous, then classical tie-break mechanisms can be adopted to prioritize event processing and to avoid permanent blocking situations.

Further, the guarantee of freedom from the domino effect within the rollback scheme can be provided in ECS by integrating classical sparse state saving strategies (see, e.g., [15–21]), aimed at (infrequently) logging the state image of the simulation objects (either for recoverability purposes within the synchronization scheme or for on-the-fly/a-posteriori inspection of the state trajectory of the simulation objects) with forced logs taken so to avoid any rendez-vous event to be reprocessed as an artifact of the reconstruction phase of non-logged state images. Essentially this implies forcing additional logs on both source and destination objects right after the rendez-vous is processed.

We note that in case each simulation object only accesses its own state (namely the data structure logically associated with such a state), which is the only scenario valid for classical PDES programming approaches, then the performance penalty incurred by ECS compared to classical optimistic PDES synchronization is exclusively related to opening the access into the sibling page tables upon dispatching some object along a given worker thread. However, thanks to the current support offered by modern processors for fast system-call paths, the corresponding `ioctl` operation imposes negligible overhead except

for simulation models with very fine event-granularity. On the other hand, the management of rendez-vous events induces a cost to be necessarily paid to allow the parallel run to mimic a classical sequential one where the simulation events are processed in non-decreasing timestamp order, while jointly being allowed to access any valid memory location belonging to the state of the simulation model (namely any memory location logically belonging to the state of some object included in the simulation model). But, as we will show in the next section, this has the potential for extremely simplifying the application programmer job.

4 Programming the Multi-Agent Exploration Model

To assess the programmability and performance of ECS, we have implemented a distributed multi-robot exploration and mapping simulation model, according to the results in [22]. Specifically, a group of robots is set out into an unknown space, with the goal of fully exploring it, while acquiring data from sensors (e.g., cameras, lasers, ...) which are used to map the environment. The robots are equipped with enough processing power to elaborate the sensors data online (thus, the map is constructed during the exploration), so as to allow them to rely on the acquired knowledge to drive the exploration in a more efficient way. Specifically, 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 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. Whenever two robots enter a proximity region, they perform three different actions: i) they use their sensors to estimate their mutual physical position—recall that they are just in *proximity*; ii) they verify the goodness of their position hypothesis by creating a rendez-vous point in the explored part of the region, and trying to meet again there; iii) if the hypothesis is verified, they exchange the data acquired during the exploration, thus reducing the exploration time and allowing for a more accurate decision of the actions to be taken. Additionally, in case step ii) succeeds (i.e., the robots actually meet in the rendez-vous point), it means that the estimation of their respective position is correct. Therefore, they can form a *cluster*, i.e. they can start exploring the environment in a collaborative way. Specifically, this collaborative exploration can take place in two different ways. On the one hand, they jointly define (by relying on *cost* and *utility* functions, as defined in [22]) their next exploration targets, so that they can minimize the time required for a complete environment exploration. On the other hand, they might decide to make a *guess* about the position of other robots (the total number of which is known) which are not part of the cluster yet. In the latter case, one of the robots (the one for which the utility/cost ratio is convenient) targets the hypothesized position. If a robot is found there, the aforementioned steps are carried out, so as to increase the knowledge of the environment.

When implementing a PDES simulation model for this scenario, three main hindrances are found. First, discovering the presence of a nearby robot can be difficult. In fact, either the robots must communicate to each other their current position (thus exponentially increasing the number of exchanged messages, which in turn can limit the performance of the simulation), or they have to notify it to specific simulation objects (i.e., the regions), again increasing the number of messages exchanged. Second, to estimate the respective position of the agents, many simulation events could be required. In this specific case, these events should be marked with the same timestamp, thus requiring efficient (but non-negligible in cost) tie-breaking approaches, like the one in [23]. Third, exchanging map information could entail a data transfer non-negligible in size, posing a huge burden on the communication subsystem. Additionally, all these programmatic steps are not straightforward, as they force the modeler to reason according to the state-separation paradigm proper of PDES.

On the other hand, as mentioned, relying on ECS allows for a completely transparent synchronization of the simulation objects involved in any mutual state update, which therefore simplifies the development process of the simulation model. In our implementation[1], we rely on two different types of simulation objects, namely active ones (implementing the robots) and passive ones (implementing regions of the exploration environment). More specifically, the environment is represented as a square region, divided into hexagonal cells. This choice allows us to define a meaningful mobility model for the agents, and at the same time allows us to define proximity regions which are used by the agents to detect the presence of other robots in the nearby. Also, in our model, periodic events occurring into any cell are envisaged as the basis for modeling the evolution (inside the cell) of any phenomenon characterizing the dynamic change in the state of the explored region.

At simulation startup, each passive simulation object creates random obstacles (which prevent the agents from reaching any neighbour cell), mimicking a rescue scenario where an open space is modified by an accident and the robots are used to explore it for rescue activities. At the same time, each passive object instantiates in its private simulation state (by relying on a traditional `malloc` call) a *presence bitmap*. Each bit is associated with a specific robot, and its value is associated with the robot being in the cell or not. By relying on a fast bitmap scan, each robot (thus each active simulation object) is able to discover which ones are present in the cell. Finally, the passive object registers its simulation state by storing a pointer in a global array called `states[]`, thus allowing any other simulation object to directly access (and/or modify) it. This is done by relying on standard (sequential) code, where the modeler is not required to rely on any platform-specific API, as illustrated in the following code snippet which is executed upon initializing the simulation model.

```
1 // Allocated state
2 state = malloc(sizeof(agent_state_type));
```

[1] The full source code of our model implementation can be found at `http://svn.dis.uniroma1.it/svn/hpdcs/root_sim/trunk/examples/robot_explore/`.

```
3
4  for(i = 0; i < 6; i++) {
5          if(isValidNeighbour(me, i)) {
6                  // With a random probability, an obstacle prevents from getting there
7                  if(Random() < OBSTACLE_PROB) {
8                          state->neighbours[i] = -1;
9                  } else {
10                         state->neighbours[i] = GetNeighbourId(me, i);
11                 }
12         } else {
13                 state->neighbours[i] = -1;
14         }
15 }
16 // Allocate the presence bitmap
17 state->agents = malloc(BITMAP_SIZE(n_prc_tot - num_cells));
18 bzero(state->agents, BITMAP_SIZE(n_prc_tot - num_cells));
19 // Register the state
20 states[me] = state;
```

`isValidNeighbour()` is a model-specified function determining whether a cell is placed on the boundary of the square region, `GetNeighbourId()` is a model-defined function which performs hexagonal-to-linear coordinates conversion for detecting a neighbour id, `num_cells` is a variable defined by the model (and initialized at simulation startup by the user) which tells how many cells must be used to represent the square region, `BITMAP_SIZE` is a model-defined macro which converts the number of agents to be represented into the bitmap to a number of sequential bytes providing the relative number of bits, `n_prc_tot` is a variable initialized at simulation startup which tells the total number of simulation objects in the current run, and `me` is a unique integer used to identify a specific simulation object, passed as input to callbacks giving control to the application for event processing (as in most traditional simulation frameworks). At this point, when an active agent executes the event associated with the entrance within a specific hexagonal cell (the id of the cell is piggy-backed on the event), registering its presence in the cell is as simple as:

```
21 state->current_cell = event_content->cell;
22
23 // Register the position of the robot in the cell
24 cell = (cell_state_type *)states[state->current_cell];
25 cell->present_agents++;
26 SET_BIT(cell->agents, me - num_cells);
```

Then, the agent has to acquire information about the environment (which in our model is represented by the obstacles in the current cell) and has to detect the presence of additional robots. Since we have direct access to the cell's state thanks to ECS, this can be easily done by reading this information from it:

```
27 // Mark the cell as explored and "discover" the surroundings
28 state->visit_map[state->current_cell].visited = true;
29 memcpy(&state->visit_map[state->current_cell].neighbours, cell->neighbours, sizeof(unsigned
       int) * 6);
30
31 // Is there any other robot in the cell?
32 if(cell->present_agents > 1) {
33         <scan the bitmap>
34 }
```

In case one robot is found in the cell, then the agent simply "merges" its view of the environment. This can be easily done by relying again on the `states[]` array, provided that each robot registers a pointer to its private state in it at simulation startup:

```
35 robot = (agent_state_type *)states[robot_index];
36 for(j = 0; j < num_cells; j++) {
37         if(robot->visit_map[j].visited) {
38                 memcpy(&state->visit_map[j], &robot->visit_map[j], sizeof(map_t));
39         } else if (state->visit_map[j].visited) {
40                 memcpy(&robot->visit_map[j], &state->visit_map[j], sizeof(map_t));
41         }
42 }
```

No notion of parallelism is present in the shown code snippets from the simulation model, yet (by relying on ECS) ROOT-Sim is able to run the simulation model in parallel, exploiting the computing power offered by multi/many-core architectures. We emphasize that the simulation model is written in a pure sequential style, without the need for respecting any programming model constraint, except for the signature of the event handlers, which is compliant with the one adopted by ROOT-Sim. Therefore, by relying on ECS, any sequential model developed for the traditional DES paradigm can be easily run concurrently, by making extremely minor modifications to it.

As a final note, ECS is triggered in different lines of the above code snippets. In particular, lines 25 and 32 trigger the synchronization with a cell (i.e., the current cell hosting the robot), while lines 37, 38, and 40 synchronize with another robot, namely one of the (possibly multiple) robots found to be in the cell. It is interesting to note that lines 25 and 32 access the state in different ways, the former being a read/write operation, the latter being simply a read operation. Similarly, line 37 accesses the state in read mode, while lines 38 and 40 entail writing multiple bytes of memory. Nevertheless, despite the multiple accesses, ECS requires executing the synchronization protocol only once per each other simulation object the state of which is being accessed while processing the current event. It is interesting to note that, if the same model were implemented using traditional message passing, multiple messages should have been exchanged to support the execution of a single event.

5 Performance Data

In this section we report performance results related to the execution of the previously described simulation model. The hardware architecture used for running the experiments is a 64-bit NUMA machine, namely an HP ProLiant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12MB L3 cache (6 MB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system is 64-bit Debian 6, with Linux Kernel version 2.6.32.5 (with supports for ECS [7]). The compiling and linking tools used are `gcc` 4.3.4 and binutils (`as` and `ld`) 2.20.0.

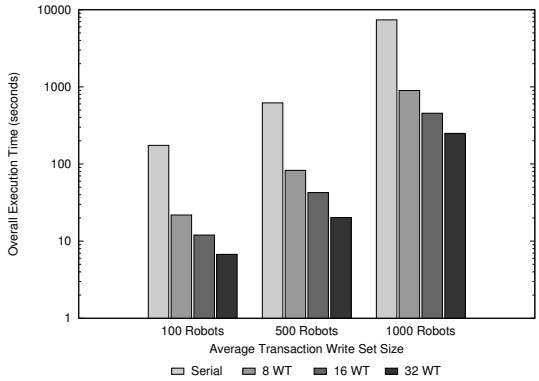

Fig. 2. Sequential vs parallel ECS-based execution times.

We simulated a region with 4096 cells, and we varied the number of agent (robot) units moving around between 100 and 1000, passing though the intermediate value of 500. The higher values (say 500 and 1000 agents) give rise to average agent density into the explored space on the order of 0.12 and 0.24 per cell, respectively. Although these values might be above realistic settings, especially for cells modeling regions with non-minimal size, they help anyway assessing how the performance of the ECS support for easy of programming and transparent parallelization scales vs variations of the ratio between the number of passive simulation objects (the cells) and the number of active ones (the agents). On the other hand, the configuration with 100 agents gives rise to an average density of per-cell agents of 0.02, realistically representing cases where, e.g., a reduced number of highly specialized agents is employed for the exploration of a non-minimally sized region.

In Figure 2 we report the execution time of the simulation model for the three different cases, and for different amounts of ROOT-Sim worker threads (say 8, 16, and 32) deployed on top of the multi-core computing system. We also report the execution time for running the very same application code sequentially, on top of a classical calendar-queue scheduler. By the data we see how the parallel runs provide speedup that ranges between 30 and 35, which also linearly scales while varying the number of worker threads in the parallel simulation platform. This demonstrates the effectiveness of ECS in delivering adequate performance, beyond providing supports for easy of programming. Also, the implementation patterns used in our model can constitute a reference for different scenarios entailing both passive (region) and active (agent) simulation objects.

6 Conclusions

In this article we have discussed how to exploit the innovative ECS (Event and Cross State) synchronization protocol for discrete event simulation in order to easily implement a multi-agent exploration model. The objective of ECS is to

provide the illusion of a sequential style execution mode for models based on multiple simulation objects that develop (mutual) cross-state dependencies via direct cross-references on their states, while actually running the model in parallel. With ECS, the programmer is exposed to a classical and easy sequential style approach for coding the state transitions associated with the occurrence of simulation events (e.g. the application code is allowed to reference any valid memory location while performing whichever state transition, under the programmer's illusion that all the transitions are sequentialized). On the other hand, the ECS-based run-time environment, which has been integrated within the ROOT-Sim open source simulation platform, allows for executing the objects (namely, the simulation events destined to them) concurrently, while correctly maintaining causality across state transitions. We have discussed the relation between ECS and multi-agent exploration models from a twofold perspective: on one side we have shown code snippets illustrating the simplicity according to which the programmer can code his model; on the other side we have provided data related to speedup results achieved via ECS-based runs (which are able to transparently support parallel execution of the sequentially conceived code) on top of a 32-core commodity machine.

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: RoboCup 2001: Robot Soccer World Cup V, Springer-Verlag (2002) 102–111
2. Di Sanzo, P., Antonacci, F., Ciciani, B., Palmieri, R., Pellegrini, A., Peluso, S., Quaglia, F., Rughetti, D., Vitali, R.: A framework for high performance simulation of transactional data grid platforms. In: Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques. SimuTools '13, ICST, Brussels, Belgium, Belgium, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2013) 63–72
3. Fujimoto, R.M.: Parallel discrete event simulation. Communications of the ACM **33**(10) (October 1990) 30–53
4. Low, M.Y.H., Gan, B.P., Wei, J., Wang, X., Turner, S.J., Cai, W.: Shared state synchronization for HLA-based distributed simulation. Simulation **82**(8) (August 2006) 511–521
5. Gan, B.P., Low, M., Wei, J., Wang, X., Turner, S., Cai, W.: Synchronization and management of shared state in HLA-based distributed simulation. In: Proceedings of the Winter Simulation Conference. (December 2003) 847–854
6. Pellegrini, A., Vitali, R., Peluso, S., Quaglia, F.: Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In: Proceedings 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. MASCOTS, IEEE Computer Society (August 2012) 134–141
7. Pellegrini, A., Quaglia, F.: Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies. In: Proceedings of the 2014 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. PADS, ACM (May 2014) 105–116

8. Quaglia, F., Pellegrini, A., Vitali, R., Peluso, S., Didona, D., Castellari, G., Gheri, V., Cucuzzo, D., D'Alessio, S., Santoro, T.: ROOT-Sim: The ROme OpTimistic Simulator - v 0.99 RC-1. `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/` (October 2011)
9. Popov, K., Vlassov, V., Rafea, M., Holmgren, F., Brand, P., Haridi, S.: Parallel agent-based simulation on a cluster of workstations. In: Euro-Par. (2003) 470–480
10. The Mozart Programming System. `http://mozart.github.io/`
11. Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., Spagnuolo, C.: A framework for distributing agent-based simulations. In: Euro-Par Workshops (1). (2011) 460–470
12. Hybinette, M., Kraemer, E., Xiong, Y., Matthews, G., Ahmed, J.: Sassy: A design for a scalable agent-based simulation system using a distributed discrete event infrastructure. In: Proceedings of the 2006 Winter Simulation Conference. WSC, Society for Computer Simulation (2006) 926–933
13. Richmond, P., Walker, D.C., Coakley, S., Romano, D.M.: High performance cellular level agent-based simulation with FLAME for the GPU. Briefings in Bioinformatics **11**(3) (2010) 334–347
14. Marurngsith, W., Mongkolsin, Y.: Creating gpu-enabled agent-based simulations using a pdes tool. In Omatu, S., Neves, J., Rodriguez, J.M.C., Paz Santana, J.F., Gonzalez, S.R., eds.: Distributed Computing and Artificial Intelligence. Volume 217 of Advances in Intelligent Systems and Computing. Springer International Publishing (2013) 227–234
15. Fleischmann, J., Wilsey, P.A.: Comparative analysis of periodic state saving techniques in time warp simulators. In: Proceedings of the 9th Workshop on Parallel and Distributed Simulation, IEEE Computer Society (June 1995) 50–58
16. Preiss, B.R., Loucks, W.M., MacIntyre, D.: Effects of the checkpoint interval on time and space in Time Warp. ACM Transactions on Modeling and Computer Simulation **4**(3) (July 1994) 223–253
17. Quaglia, F.: Combining periodic and probabilistic checkpointing in optimistic simulation. In: Proceedings of the 13th workshop on Parallel and distributed simulation, IEEE Computer Society (1999) 109–116
18. Quaglia, F.: Event history based sparse state saving in time warp. In: Proceedings of the 12th Workshop on Parallel and Distributed Simulation, IEEE Computer Society (1998) 72–79
19. Quaglia, F.: A cost model for selecting checkpoint positions in Time Warp parallel simulation. IEEE Transactions on Parallel and Distributed Systems **12**(4) (February 2001) 346–362
20. Rönngren, R., Ayani, R.: Adaptive checkpointing in Time Warp. In: Proceedings of the Workshop on Parallel and Distributed Simulation, Society for Computer Simulation (July 1994) 110–117
21. Cucuzzo, D., D'Alessio, S., Quaglia, F., Romano, P.: A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. In: Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, Los Alamitos, CA, USA, IEEE Computer Society (2007) 227–234
22. Fox, D., Ko, J., Konolige, K., Limketkai, B., Schulz, D., Stewart, B.: Distributed multirobot exploration and mapping. Proceedings of the IEEE **94**(7) (July 2006) 1325–1339
23. Mehl, H.: A deterministic tie-breaking scheme for sequential and distributed simulation. In: Proceedings of the Workshop on Parallel and Distributed Simulation, ACM (1992)