SAPIENZA
UNIVERSITÀ DI ROMA

Faculty of Computer Engineering
Computer Science and Statistics

Master of Science in Engineering
in Computer Science

Master Thesis

Dynamic clustering of simulation objects in speculative parallel simulation systems

Academic Advisor
Prof. Francesco Quaglia
Dr. Alessandro Pellegrini

Candidate
Nazzareno Marziale

Academic Year 2014/2015

"Truth is ever to be found in simplicity,

and not in the multiplicity and confusion of things."

—Isaac Newton

Contents

List of Figures

Chapter 1

Introduction

In the field of simulation, the Discrete Event Simulation (DES) is the process of analysing the behaviour of complex system, created by an analyses of real word, and realizes a sequence of ordered events that describes it. In this system, events describe how the simulation has to change its state in a specific point in time. The simulation is defined discrete because proceeds forward in time from an event at the time of the next event

Since the amount of data to be analysed and the complexity of Simulations increases more and more, it appeals to a solution in parallel and distributed system, because many problems usually have a good part parallelizable. This solution takes the name of Parallel Discrete Event Simulation (PDES). The execution of the simulation consists of processing the discrete events, as described above, by the Logical Processes.
A *Logical Process* is the representation of physical actor inside the simula-

tion, that has a state and it changes its status processing the events.
To ensure proper output of the simulation, we need to define some rules of synchronization between all simulation's components. The main constrain is the local causality, according to this role the events of a Logical Process must be processed in timestamp order. If this constraint is satisfied, the parallel simulation will have the same results than sequential. The synchronization algorithm in which attention will be focused in the following thesis is *optimistic*, under which the bond of causality may be violated, but as soon as it is detected is made rollback to return in a consistent execution state. In order to minimize the number of rollback and try to execute the simulation in a sequential way, each worker thread schedule the Logical Process with the minimum timestamp over all the Logical Processes available for this worker thread.

Given the increasing diffusion of shared memories, and therefore the possibility to access the status of the different Logical Processes, at the level simulation has been necessary to make access to different states transparent to the programmer level. This specific part is discussed in the thesis of my colleague Francesco Nobilia, he makes the system able to track accesses to memory areas that not belonging to the specific Logical Process.
Under this assumption, in this thesis we deal with the problem of creating a Group of Logical Process. A Group is a set of Logical Processes running in a serial manner and each of them can accesses in a safe way to the memory of others. This is assured since each group, so each Logical Process within the group, is scheduled exactly from one Worker Thread. The Worker threads

when it decides to schedule the group performs two operations:

1. Access to the message queue of each Logical Process and extracts the minimum.

2. Of all the minimum extracted from various queues it is taken the least.

This policy ensures that the execution of Group of Logical Processes is both serial and does not violate any causal connection.
Therefore we deals with the problem of creating a Group of Logical Processes, then such Local Processes must belong to that particular group. In order to understand the interconnection between different Logical Processes, we exploit the cross-state dependencies to build an access statistics, through which the simulation engine will determine the conformation of groups.
As well as previously introduced, each memory access of other Logical Processes units within the group is not considered as a Event Cross State, this involves a significant increase in performance since the Logical Processes do not have to synchronize with each other. Since the system we consider is optimistic, in this thesis, we analyse also the problem of frequent rollback. With the our solution, we can found a relationship between LPs, creating a symbiotic execution, avoiding rollback. This leads to create a new kind of rollback, in fact, when a process that is located within a group must perform a rollback, this means that every other processes that belong to the group perform rollback at the appointed time.
At the end of a period ($group\_era$), groups are destroyed and the system continues to collect statistics of the interactions between the Logical Processes

and than we can create a new configuration of the groups, in order to adapt the system to the simulation's evolution.

This solution has been implemented inside the ROme OpTimistic Simulator (ROOT-Sim).

The rest of this thesis is organized as follow: Chapter 2 presents all works related to the one discussed in the thesis and the different implementations developed to solve this problem. Chapter 3, we explain more in detail the simulator on which the solution is implemented. The work conducted in this thesis is explained thoroughly in Chapter 4 and Chapter 5 we expose the results of tests and the improvements observed. Finally, in Chapter 6 we explain the conclusions that have been reached.

Chapter 2

Background on Optimistic Simulation

In recent years, parallel computing has been a topic of great interest in research. Resource sharing hardware can perform with greater computational power, allowing an increased performance, in a way which is completely transparent to the user. Indeed, in the multicore devices is big theme during the 2004, because if the Moore's Law predicts an exponential grow in speed up of CPU, this does not continue forever owing to the hardware physical restriction.

In this situation, to improve the performance we have to add more cores, in this way we can execute different operations simultaneously. The multicore approaches leads concurrency problems. In detail, a multi-thread program has to taking into account the coherency of the data structure and the syn-

chronization phase. However, if exist in our program different control flow that could be executed in parallel, the speed-up of execution is significant.
The rapid progress of construction at the base of these virtual systems arises from the need to run simulations on a large scale and in the most diverse fields: engineering, economics, military research, biology and science in general (see, e.g. , [1], [2] and [3]). Indeed, we can study and/or interact with the behaviour of complex systems during their evolution (*symbiotic simulation*) or before their actual realization (*what-if analysis*). Simulations of this type require, for their processing, massive computing resources that can only be obtained in parallel and distributed environments, where the various nodes cooperate to achieve a common result, subject to stringent requirements for consistency and synchronization, reliable communication latency.

The survey in the field of parallel and distributed simulation begins in 1979 with the article by Chandy and Misra in [4].
The concept of PDES (Parallel Discrete Event Simulation), described for the first time in [5], is an evolution of the previous DES (Discrete Event Simulation). It consists in a distributed paradigm for the execution of discrete-event simulation models.By discrete-event simulation we mean an arithmetic-logic model capable of representing physical systems in the real word, schematized through algorithms and / or mathematical formulas. Each simulation model is associated with a state that represents the totality of information managed by the application, and a set of discrete events generated during the evolution of the model, which lead to changes in the system state.
A simulation is called discrete when the operations associated with the events

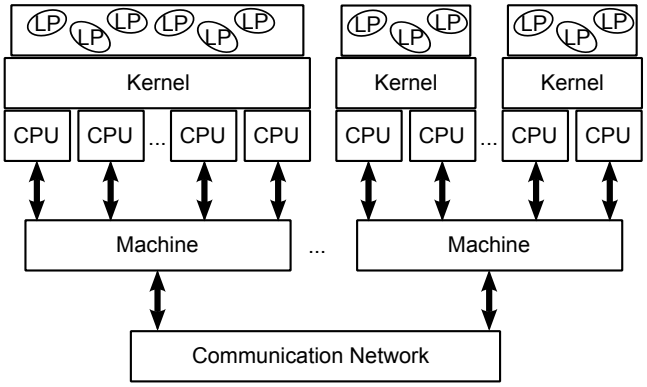

Fig. 2.1: Parallel Discrete Event Simulation

take place instantly and have a impulsive duration. The idea behind PDES, whose architecture is shown in Figure 2.1, is to running a simulation program on parallel (remote or local) computers and based on the processing of discrete events. Each event can produce changes, more or less complex, on portions of the simulation state. The events are correlated in time by logical discrete time called timestamp, when an LP executes an event, it updates its current runtime (Local Virtual Time, LVT).
This dependency makes it possible a form of synchronization and coordination between all processes participating in the simulation, so as to achieve a common and correct result.

PDES can be viewed as a set of N objects, called Logical Process (LP), indicated with $LP_0, LP_1, \ldots, LP_{N-1}$, each of which is associated with a state, $S_i \subseteq S$, containing a subset of variables that are strictly necessary to the evolution of single instance of the simulation. The set S is the global simulation state and then keeps all information relevant to the simulation. During a simulation, the LP performs two main operations:

1. **Event processing:** this leads to the advancement of the logical simulation time, changing at the same time the LP state and thus the simulation state. Events may have been generated by the LP itself or by other LPs.

2. **Event generation:** during the processing of an event, the LP may decide to send an event, either to himself or to another LP.

Through these basic operations an LP can interact with other LPS in order to carry out the simulation, to achieve the desired result. PDES, being a parallel simulation, does not ensure that the execution of messages is sequential. Indeed in a concurrent simulator, one of the major problems is to always ensure the *state consistency*, so the global state of the system must be always consistent with the model specification.
In this way, we are sure that there are no erroneous transitions that may affect the final simulation result.
In order to ensure state consistency, we are presented two different synchronization paradigms.

2.1 Conservative vs Optimistic Simulation

There are several strategies to implement PDES consistency, wiich are widely discussed in the literature ([6] [7]). The three main ones are: *conservative*, *optimistic* and *hybrid*. The first to be introduced was the conservative ([8] and [9]). This strategy avoids the occurrence of causality errors at runtime,

since the execution of an event takes place only if it is guaranteed to give a correct effect on the LP state, meaning that the event to be processed is *safe*. In detail, conservative simulation requires that an event e_1, with timestamp T_1, is processed only if there is no other event e_2 with timestamp less than T_1 or if the system is able to determine that it is impossible to get another event e_2 with timestamp $T_2 < T_1$. With these assumptions, we are sure that execution proceeds without any violation of causality and that the state of the LP will be always consistent.

To determine if an LP can process an event in the future in a safe manner we can resort to the *lookhead*: if an LP can assert during the execution of event e_1 that up to time $T_1 + \Delta$ all events processed will be safe, we say that the process has a *lookhead* equal to Δ.

In detail, Δ is the smallest increment of time according to which new events are generated, so it is a quantity associated to the specific model. In some cases Δ can also be zero.

This solution, however, does not fully exploited the parallelism offered by the underlying hardware, since even if two events e_1 and e_2, which are located on two different LP, are not logically dependent, the system might force a sequential execution even if it is not necessary.

The approach that we will examine in this thesis is the *optimistic* one. Differently from the conservative solution, optimistic synchronization selects looking only at the local LP, without taking into account the causal dependency with others LPs.

The most significant example of optimistic synchronization is the *Time Warp*

protocol in [10]. In this approach events are processed as soon as they are available,and if later a causality error is detected, the system is returned to a consistent state, from which it can process the new event that caused the error. This operation is called *rollback*, which we analyse in detail in section 2.3. With this solution the available parallelism is fully exploited because it does not carry out audits of other LPs to see if an event is safe.

One of the main studies on the Time Warp, and how this is applied to the DES platform, is led by Jefferson in [11]. Indeed this article explains how the Time Warp Operating System (TWOS) has been realized and discusses its performance.

TWOS is a system that exploits several processors to increase the parallelism of simulation. Moreover, the main innovation that has been presented in [10] is the possibility to *rollback* and then it can exploit the whole concept of running the simulation optimistically.

Another important study in this area was conducted and presented in the work [12], including a proposal for global scheduling mechanism, that we analyse in Section 2.2. It also introduces the presence of a distributed queue of events to improve the performance and load sharing for Logical Processes, where each one of these is taken over by a core.

2.2 Global Virtual Time

Another important concept associated with the rollback is the Global Virtual Time (GVT), described in [10]. In PDES it is important to determine a certain instant of time at which all virtual LPs arrived, so that we can consider all the events that occurred before this instant as committed. Indeed, if all the Logical Processes have processed correctly the events with at a timestamp bigger than x, where x is the minimum time between all the latest processed events,no LP may send an event to another LP with a timestamp less than x. In this way we can ensure that no LP will perform a rollback to a Local Virtual Time less than x.

So we can define the Global Virtual Time as the smallest LVT value t among all the LPs at a given Wall Clock Time instant, and as the minimum virtual time of all messages that are not received in the system at real time t.

There are different implementations and definitions of GVT, such as [13], [14], [15], [16], [17]. However, the computation of GVT in PDES system is critical because it allows to establish a time frame before which all events can be considered *committed*. This allows us also to delete all the information necessary for rollback before this time, in order to optimize the use of memory and avoid keeping unnecessary information for the system. This operation is called *fossil collection*.

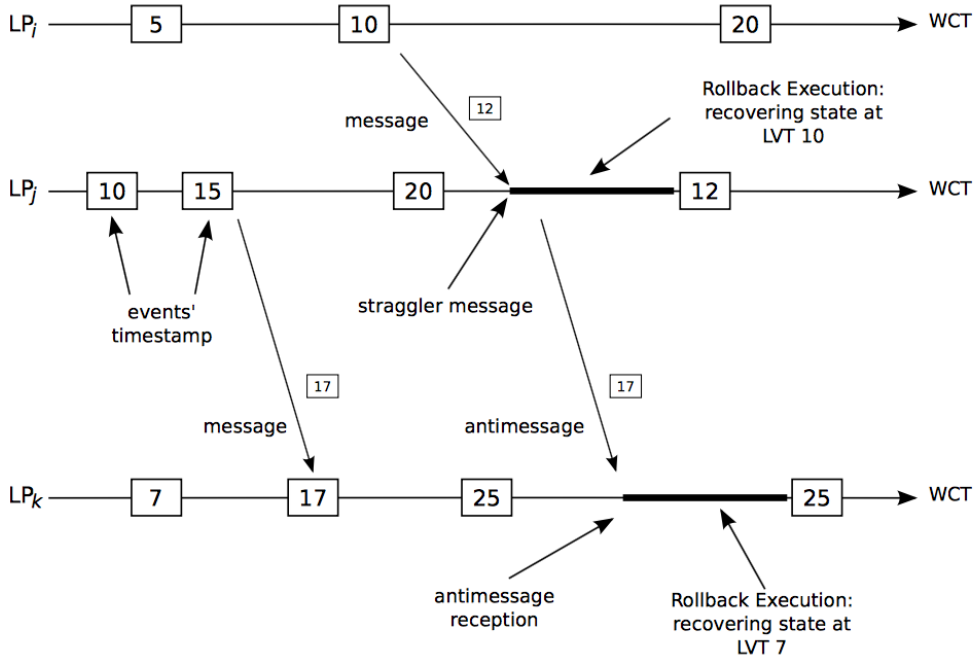

Fig. 2.2: Rollback Example

2.3 Rollback Approaches

In optimistic simulations, as described above, one of the major challenges is how to perform a *rollback*. Indeed, it can happen that a LP, which is at a time x, receives a message from another LP at time y where $x > y$ (*straggler message*). In order to process this new event, the LP has to go back to a time less than or equal to y, and then it has to realign its state at the time of execution. This is because being a optimistic simulation, LPs run events in a speculative way in order to try to optimize the simulation performance. Since the rollback operations can affect the simulation's performance significantly, different solutions have been developed in order to reduce its overhead. Now, we start analysing how we can compute the rollback in order to ensure

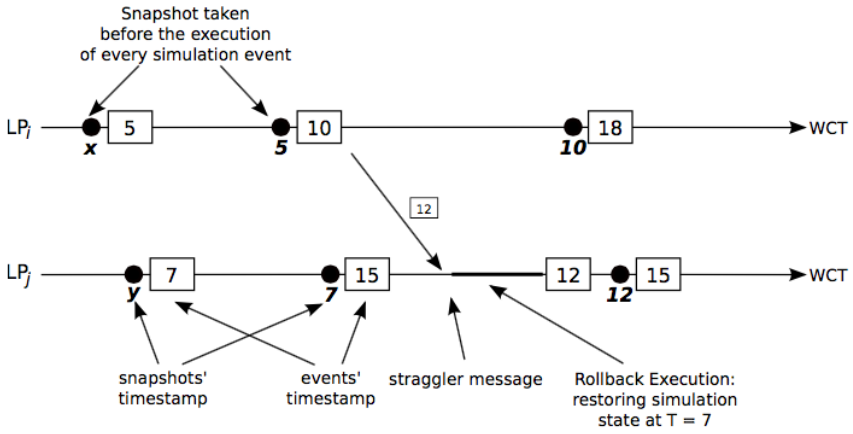

Fig. 2.3: Copy State Saving

the correctness of the execution, and since it has an high cost, we will try to optimize its performance.

2.3.1 State Saving

In this approach, to ensure that it is possible to rollback an LP's state, the system stores the old value and when a causality violation is detected, the correct state is restored by coping back the saved value. We show some technique to realized rollback with State Saving approach, in order to reduce the cost of this operation.

2.3.1.1 Copy State Saving

The problem of bringing LPs back to a consistent state is introduced into the work of Jefferson [10], which proposes a very simple solution. Whenever an

LP processes an event, exactly before the simulation engine takes a snapshot of LP's state and stores in a queue of states (Figure 2.3).

As soon as an LP receives a straggler message, the simulator restores the state immediately before the time of the straggler message and sends *antimessages* for all messages with timestamp bigger than the restore point.

An *antimessage* is a negative copy of a message that tells the system that the positive copy must no longer be processed, the negative copy annihilates the positive one.

This solution leads to excessive memory usage to save the state after that every single event is processed.

The only solution in this implementation to reduce the space used is to run frequently the GVT reduction, but this leads, as previously described, to performance reduction.

2.3.1.2 Sparse State Saving

To optimize state saving so as to reduce the impact of CSS, different approaches have been developed, which all fall under the name of Sparse State Saving (SSS) [18] [19] [20].

With SSS, a checkpoint is not taken before the execution of each event, but *sparsely*. SSS is divided into two main categories: Periodic State Saving (PSS) and Adaptive State Saving (ASS). In the first, the checkpointing interval is determined in a static manner, while in the second the period varies depending on the simulation model's dynamics.

With this methodology, consequently, the rollback mode changes as well. The

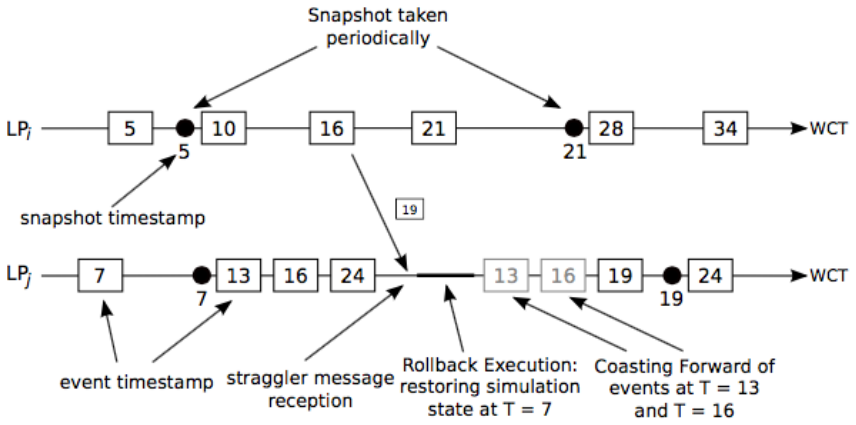

Fig. 2.4: Sparse State Saving

following two cases may occur:

1. The case similar to the approach above, in which we have a state with a Local Virtual Time equal to the time at with we have to roll back. This is the most simple situation, indeed, we restore the LP's state with the copy and we send antimessages of all subsequent messages.
   $LVT_{snapshot} = LVT_{rollback}$.

2. The state that was saved has a Local Virtual Time smaller than the point at which the LP has to go back, because the Local Virtual Time of the straggler message is different from LVT of the state (Figure 2.4).
   $LVT_{snapshot} < LVT_{rollback}$.

In the second case, to restore a consistent state, we should rerun all events that are present between the point at which the system has taken the snapshot, and the point at which it is necessary to rollbak. These events can not

be executed normally because otherwise it could be introduce into the simulation a sequence of duplicate events, which would hamper the correctness.

The technique used in this case is called *Coasting Forward* [20]. The events are replayed up to the rollback point, without sending messages to other LPs. This execution is called *Silent Execution* and is required to bring the LP exactly at the right LVT to run the Straggler Event with a correct state, starting from the first available simulation snapshot.
In this way we optimize the use of storage, if the checkpointing interval is chosen appropriately. Indeed, if it is too short, we don't get any benefit in memory usage. In the other case, if it is very large, the simulator has re-run many events silently. The optimal checkpointing interval [21] can be computed as:

$$\chi_{opt} = \left\lceil \sqrt{\frac{2\delta_s}{\delta_c} + \left(\frac{N}{k_r} + \gamma - 1\right)} \right\rceil \tag{2.1}$$

where:

δ_s is the average time to take a state snapshot;

δ_c is the average time to execute the coasting forward operation;

N is the total number of committed events;

k_r is the number of rollbacks executed;

γ is the average rollback length;

2.3.1.3 Incremental State Saving

The previous solution does not take into account the size of the checkpoint and the time required to pack it. In fact, if the LP's state is very large, the

saving operation can be costly. In [22] and [23] a first version of Incremental Save State are presented.
These solutions limit the amount of memory usage and the time spent to pack the checkpoint. The problem with these implementations is that they are not transparent to the user, indeed the latter must be aware of the rollback operation and state saving, so he has to manage state changing in such a way that after the system can perform the rollback.

Subsequently, in [24], thanks to the instrumentation of software, it is possible to make this operation transparent to the programmer. Indeed, the assembly code of the model is parsed and all transitions that update the state are replaced by a particular function call, that before updating the value, saves a copy.

In conclusion, the works in [25] and [26] provide an approach completely transparent to the user, using again the instrumentation of the code, but by inserting an intermediate level that helps to determine which areas of memory have been modified, without saving information that has not been changed. This is implemented using a bitmap and setting to 1 the bits associated with memory areas modified during the execution of events.

2.3.2 Reverse Computing

Reverse computing is a different technique to realize the rollback operation, indeed in this case the system does not store the state information, but

exploits revers events which are able to undo the effects of events processed in forward mode.

In [27] and [28] two solutions are presented, both exploiting the instrumentation of the code.Indeed in the first solution, the model code is manipulated by a compiler that produces two different outputs. The first one is the model code with some additional instructions for the execution of reverse computing. While the second output is reversing code, necessary when an LP receives a straggler message and it has to perform a rollback.
The second solution uses the LLVM framework [29] to store the information about messages that alter the state. Indeed, LORAIN makes the key assumption that only instructions that affect memory have the capacity to alter state. This framework allows to the system to manage the *destructive* instruction.
Some example of *destructive* instruction is variable assignment or division, because after the execution of this operations, we can not compute the original value through the inverse.
LORAIN analyses the model code looking for `store` instructions and augments the `message` structure with the LLVM metadata. In this way, when it is necessary to roll back, the system can use `antimessage` and restore the correct state computing the reverse of message, also if it contains destructive instructions.

Since in any case the rollback has a huge cost in terms of computing power as well as it hampers performances, our solution proposes to decrease substantially the number of rollback that must be executed. In fact, by

tracing interactions between the LPs, we can determine if there are some logical connections between LPs and then, with the new concept that we introduce of Groups of Logical Processes, we can decrease the amount of rollbacks, as we execute sequentially all the events belonging to LPs of the same group.
Furthermore, our solution is completely transparent to the user, since both the statistics of interaction and the creation of groups are executed entirely by the underling platform without requiring the programmer to modify the model code.

2.4 Grouping Simulation Objects

As regards the problem of accessing the state of another Logical Process, Fujimoto in [5] proposes a first solution. It is based on the exchange of messages between processes to access shared memory areas which are mapped to one (or more) LP's state, but the process which contains the information required could easily become a bottleneck. Also the state of process to be access must be synchronized with the LVT associated with the access operation, otherwise we're going to read either an information not yet updated or a future information, because the Logical Process has already processed other events. However, in both situations the information is inconsistent with the current simulation time instant.
An efficient approach, that was proposed, it is to duplicate the shared information on the processes that ask to access the state of other Logical Processes

but, in this case, the protocol has to deal with to maintain consistent all the informations copied in various processes. However, this solution is not transparent to the user and greatly complicates the programming model.

A solution based on the studies of Jefferson [11] is Georgia Tech Time Warp (GTW) [30]. This version is optimized to run events of small size, these types of simulation are used in the field of telecommunications, such as the simulations of wireless networks. Even in this solution, they use Logical Processes, which represent the components of simulation, and these can perform only three operations:

1. process `start` message

2. process a message received from another LP and send a new event

3. call a procedure to end the simulation process.

For what concerns shared memory between processes, GTW manages the problem during the creation of processes and messages. Indeed, each processor has a memory area which is accessed by all processes, both when the system needs to send messages and when it needs to read the messages addressed to a particular LP. This allows us to have shared memory, but requires that whenever an LP wants perform an access to send or read a message, it must take a lock. One possible solution presented is to create different pools for sending messages, but this involves different pools depending on the number of processors to which the underling platform should could messages.

In [31] the problem is dealt in a manner independent of the underlying system and the type of allocation chosen for the Logical Processes. It is proposed a solution based on the scheme State Query Time Warp (SQTW), which is an enrichment of Time Warp with the use of the State Query communication protocol (SQ). Through the SQ communication protocol every Logical Process can access the states of other LPs under the condition that data is valid, and then updated them at the same simulation time instant. To manage this type of interaction, three new messages are introduced:

1. `activate`: to activate the synchronization phase at the process P.

2. `query`: P sends the request to any other process involved in retrieving data.

3. `reply`: all other processes, once they have process the message, send it reply to the process P.

Since the simulation is optimistic, this solution is also provided with management of rollback in case where there is a causality violation. However, a solution for grouping Logical Processes is not provided, and then each time that a process must access memory of another one, it has to apply the synchronization rules even if this operation is carried out frequently.

In [32] the concept of shared state over different Logical Processes is introduced, based on the Multi-Agent System (MAS) [33] environment. Indeed, according to the configuration of this simulation environment we can not know in advance how many times and which memory areas the LPs will

want to access during their execution.
The solution proposed consists in the creation of a shared memory area, maintained by a set of additional LPs, called Communication Logical Process (CLP). All other LPs interact with this area of shared memory by sending special events to CLPs to get, update or add informations. This solution of course has the problem that the shared memory area can become a bottle-neck. This proposal does not present in any way a methodology for exclusive allocation of a specific memory portion to a selected set of Logical Processes.

In [12] and also examine the solution that is exposed for the problem that we address in this thesis. Indeed, also [12] faces the problem with the aim to decrease the amount of rollback and to decrease the amount of sent data, looking however to maintain a certain degree of parallelism. Within the article it describes a standard evolution of the state, in which there are N different Logical Processes that are running independently, and the authors introduce a new concept of Logical Process, called Extendend Logical Processes (Ex-LPs).
A Ex-LP is a set of Logical Processes, which has the ability to access in a safe way memory areas of other processes belonging to the set. Since accesses to memory areas among LPs within an Ex-LP can not be traced, the solution that has been made is to keep a list of events which occurred within the Ex-LP. On the contrary, for the accesses from external LP of the Ex-LP set, there is a split of the types of memory contained within a Ex-LP. Indeed, there are two types of memory, a *Private* that can be accessed only by the internal elements of Ex-LP, while another *Public* that can be accessed by

all. A possible implementation presented to share memory is to use software transactional memory (STM), already presented in other articles such as [34], [35], [36] and [37].
In conclusion, this work proposes a solution that does not completely fit our problem, indeed it does not address the possibility of modifying the structure of Ex-LP over time and therefore it does not allow to tune the system depending on the simulation progress.

The last work that we take into account is the work of Mehl and Hammes [38]. In this work they expose different algorithms to manage the sharing of memory over the Logical Processes in a optimistic simulator. Furthermore they assume that the rollback is performed as exposed by Jefferson in [10].
The first algorithm introduced, that exploits the rollback facility, maintains a multi-version list for each variable of single Logical Process, in this way if the LP has to roll back, it can restore a previous version of shared variable and knows which processes read and accessed to this variable. Indeed, all the Logical Processes that read a shared variable, that is after rolled back, have to rollback themselves as well.
When an LP performs a read of shared variable, the hitted object checks in multi-version list of this variable which value is coherent with the event timestamp, update the multi-version list adding this read and finally return the correct value.
On the other hand, if the LP performs an update, sends the request and if there are some reads later than this update, all the LPs that accessed this variable have to rollback at the update message timestamp. This is

mandatory because the other LPs could have read a wrong value and this update could change their behaviour.
The second algorithm uses the same concept of multi-version list, but when an LP performs a read request specify if this is read-only. When an update is performed over this variable, the owner of variable communicates, to all the LPs that accessed the information, the new value of this variable and each LP decides on its own if the rollback should be performed or not.
In the last algorithm, if an LP asks to read a shared variable, it copies the entire multi-version list locally. In this case, when the LP performs a read, it accesses the local copy, while when someone performs an update, all the local copies must be updated, and so the update message is propagated in multicast way.

In conclusion, these works propose a good solution for sharing data, but do not consider the case when this access is frequently. Consequentially we can consider to join these processes to improve the execution performance. In this way, as stated above, we decrease number of rollbacks and create a temporary relationship between several Logical Processes.

Chapter 3

ROOT-Sim: The ROme OpTimistic Simulation Kernel

In this chapter we explain the system architecture that we use and how the simulation engine works.

3.1 The Reference System Architecture

The system architecture which we use is the Symmetric Multi-Threaded optimistic simulation kernel, depicted in Figure 2.1 where we can that It is organized in different layer. On the top we can find the Logical Processes (LPs), each LP is identified by a unique number, starting from zero up to $numLP - 1$, where $numLP$ is the total number of active LPs. The second level is a simulation kernel instance and it handles different LPs.

According to the multi-threaded programming paradigm, the simulation kernel instance is divided over the available CPU cores. This is done by placing each core exactly a worker thread, thus avoiding the operating system moves the worker threads on other cores, in order to increase the simulation performance. We can also use different machines, in this case each machine has a multi-thread simulation kernel instance and all the informations are spread by exploiting a message passing-based communication network, for example MPI library [39].

The main problem multi-threaded optimistic simulation kernels is to avoid as much as possible synchronization between different LPs that are hosted on top of the same kernel instance. The data structures that during the simulation require much access are both input and output queue. Indeed, these data structures are the principal point of cross-LP dependencies because they are not only updated by the work thread that executes the LP, but also by the other worker thread to insert event in the LP's queue.
Since, we can not implement the access to these data structure using a locking mechanism because this causes scalability problems, we exploits a top/bottom-half mechanism. As shown in Figure 3.1, each LP uses an `LP_lock` to access the bottom-half queue associated in order to insert the new messages in its queue.

In order to take advantage of data locality, each worker thread can not perform all the LPs, but handles a predetermined subset of these. We can apply two different implementation of the *rebinding* phase: *static* and *dynamic*.

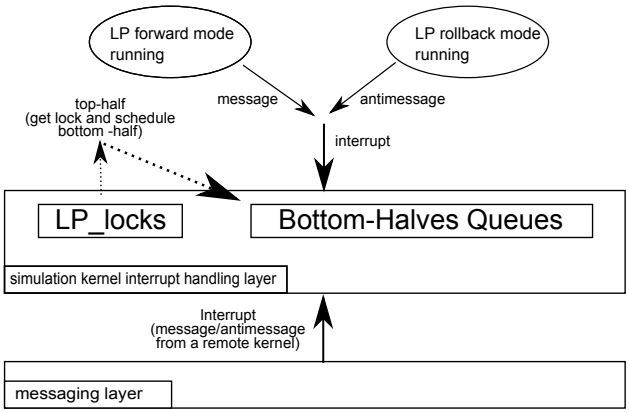

Fig. 3.1: Top/bottom halves architecture.

In the *static* solution, at the start up of the simulation, we divide all the available LPs simply assigning at each worker thread $numLP/numCore$ (where $numCore$ are the total number of cores in the machine), this assignment is fixed for the entire simulation.

The *dynamic* solution takes also into account the workload of each LP, in order to avoid that a worker thread has processes with little workload and thus either wastage computing power or proceed too far with the simulation speculatively, bringing a portion of simulation too forward in logical time. This latter case may cause a large number of rollback. To avoid these problems, the system periodically performs a rebinding in order to try to align the load among all the worker threads. Firstly, we calculate the workload for each LP (L_i) from the current LP's local virtual time (LTV_{start}) until next rebinding (LVT_{end}), taking into consideration: the number of events currently in the queue to be processed in this interval (q_i) and the average time required by

the CPU to perform an event (δ).

$$L_i = \frac{q_i \times \delta}{LTV_{end} - LTV_{start}}$$

As soon as we have calculated all the loads of LPs, we apply an *knapsack* algorithm so as to determine the ideal distribution among all worker threads. In order to avoid wasting too much time in the calculation of *rebinding*, the latter is performed after a fixed number of GVT.

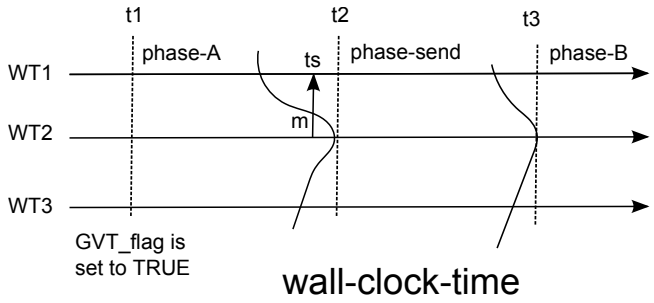

Fig. 3.2: GVT Computation phase example

As said previously, the GVT value is equal to the global minimum (across all the worker-threads) of the timestamps of messages/anti-messages that are into the event-queue.

Hence, building a GVT algorithm actually means determining some right moment for the worker thread to look at its data structures and to compute its local minimum, which will be then used for the calculation of the global minimum. We create a non-blocking algorithm to compute the GVT, in which each worker thread has to pass through different phase before to decide the final value of GVT, as shown in Figure 3.2. During the algorithm, each work thread (WT_i) computes two times the GVT, so it has two value: min_i^A and

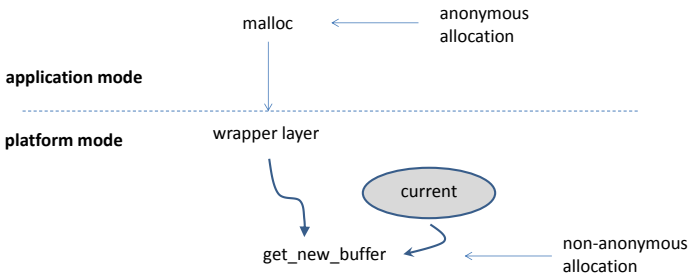

Fig. 3.3: The dual-mode execution model.

min_i^B. At the end of all the phases the WT_i proposes the $min(min_i^A, min_i^B)$ and the system determines the GVT by calculating the minimum of all the worker threads proposals.

We assume that the PDES system runs according to a dual-mode scheme where we distinguish between *application* vs *platform* modes. As soon as an simulation object is dispatched for execution from worker thread, then the system switches to application mode. A the end the control return to the worker thread that decides another LP to dispatch. To manage coherently the different kind of memory allocation, we consider a software architecture where memory allocation/deallocation services by the application code are not directly issued towards the standard malloc library. Instead, they are transparently intercepted by the underlying PDES environment and redirected to proper allocators(Figure 3.3).

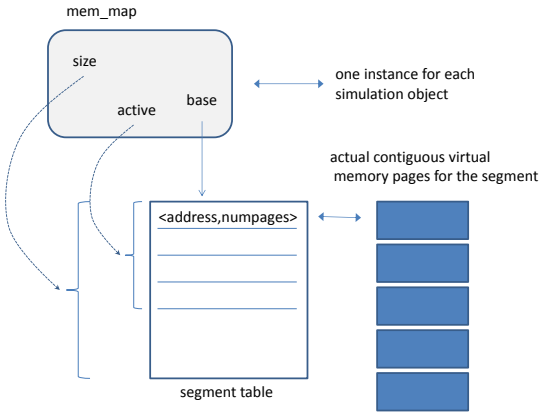

Fig. 3.4: `mem_map` data structures.

According to the above premise, whenever the application code calls a `malloc`, the underlying environment perform non-anonymous memory allocation/deallocation operations, thanks to that it can detect which is the LP that invokes the memory request.

Consequently, the function `get_new_buffer(int sobj_id, size_t size)` return different memory area depending on which simulation object performs the request. All of this operations are executed by the platform layer in a transparent way to the application. On the other hand, when the system calls memory allocation API, this is executed anonymously, in order to distinguish between application memory and platform memory.

There are different implementation of this service, we use the open-source DyMeLoR allocator [40], it allocates a large memory segment, using the classical call to malloc, and after it divides logically this memory in smaller chunk. As soon as the application needs memory, it calls the `malloc`, that is wrapped and return a number of chunks depending on the amount of memory request.

3.2 Simulation Engine

The core API of the ROOT-Sim is very simple and it consists of one call function, `ScheduleNewEvent()`, and two callback functions, `ProcessEvent()` and `OnGVT()`. The callbacks must be necessarily implemented in the simulation model to be compliant with the library. Then, the rest of the code can be implemented like a classical ANSI-C application, without any particular restriction on the use of data structures. These functions have the following signature/purpose.

`void ProcessEvent(int me, time_type now, int event_type, void *event_content, void *state)` is the callback that supports the actual processing of simulation events, and it is used by the kernel to give control to the application layer. The parameters required by this callback are: the first is `me` that represents ID of the LP being scheduled, `now` is the local virtual time of the scheduled LP, `event_type` is code, expressed with a number, associated whit the selected event, `event_content` pointers to event data structure, and finally `state` that is the state of current scheduled LP.
Inside of `ProcessEvent()` the execution is fully speculative, according to the optimistic simulation the events could be undo, but this operations are completely transparent to the programmer, indeed he has only to implements the different management of event case in order to implement the state transitions. In case of a detected inconsistency, ROOT-Sim will transparently undo the system at a coherent state, on the other hand, if no causality violation appears the simulation engine commits correct speculative events (whenever

a new GVT value is computed and the commitment horizon is moved forward).

The precessing of non-rollbackable actions are the only problem of `ProcessEvent()`. Indeed, if the programmer, use a `printf()` this could be not coherent because after the simulation could rollback at a previously virtual time and undo the `printf()`, but the output generated will not be reverted. This is a non-trivial problem associated with speculative execution, even more if transparency is enforced and the programmer is given the freedom to implement its model by relying on standard ANSI-C. ROOT-Sim offers a facility which tries to address this issue (at the cost of some delay in the materialization of the actual output).

`void ScheduleNewEvent(int receiver, time_type timestamp, int event_type, void *event_content, int event_size)` is a function that allows to schedule a new simulation event within the system, to be destined to whichever simulation object. `receiver` denotes the ID of the destination LP, `timestamp` is the logical virtual time at with the event has to be executed, `event_type`, `event_content`, and `event_size` are respectively the code of the event, the pointer to event's data structure and the size of data structure. For efficiency reasons, the invocation of this function does not immediately involve the actual deliver of the associated event to the destination LP.

Instead, events are buffered and asynchronously delivered when the execution of the current one is completed. This allows to pack together more events if the destination LP is the same, and prevents delays in the current event's

execution. We note that this asynchronous deliver does not affect the correctness of the execution, as ROOT-Sim will order events in the input queue before scheduling the next event to the destination LP. In case the delay created by this internal buffering generates an out-of-order execution at some LP, then the rollback procedure will restore consistency.

`bool OnGVT(void *snapshot, int gid)` is a callback that gives control to the application layer by also providing a committed snapshot of the simulation object. The execution of `OnGVT()` is therefore not speculative, i.e. any action taken within this function will never be undone. This means that, e.g., any I/O operation within this function is perfectly safe, and therefore it can be used to gather statistics on the ongoing simulation, if the user is aware of the synchronization strategy and does not want to rely on the facilities described in [41]. We note that, since the timestamp associated with `snapshot` refers to the committed portion of the computation, it is forbidden to call `ScheduleNewEvent()` within `OnGVT()`, because this might induce a rollback operation of already committed events. In case the user calls `ScheduleNewEvent()` in this callback, a runtime error will be generated.
`OnGVT()` additionally implements a distributed termination control: since `snapshot` is a portion S_i of the Committed and Consistent Global State (CCGS) S, according to [17] a global predicate can be locally evaluated on S_i. If the model determines that the simulation is completed for that particular LP, `OnGVT()` can return the `true` value. ROOT-Sim will collect all return values, and in case all the LPs agree, the simulation will stop.

3.3 Code Example

We present here some code snippets implementing a ROOT-Sim application which models a set of N nodes connected as a mesh, sending packets randomly to each other. The first important thing is to define the possible events handled by the model, the content of an event message, and the structure of the state:

```
1  #include <ROOT−Sim.h>
2  #define PACKET 1 // Event definition
3  #define DELAY 120
4  #define PACKETS 1000000 // Termination condition
5
6  typedef struct \_event\_content\_t {
7  time\_type sent\_at;
8  } event\_content\_t;
9  typedef struct \_lp\_state\_t{
10 int packet\_count;
11 } lp\_state\_t;
```

In this model we allow just one application-defined event, `PACKET`, which identifies the transit of a packet in the mesh. Then, we must specify the actual events' logic. `ProcessEvent()` is the only entry point for speculative event processing, so we rely on a `switch` construct to demultiplex them:

```
18 void ProcessEvent(unsigned int me, time\_type now, unsigned int event, event\_t *content,
       unsigned int size, lp\_state\_t *ptr) {
19 event\_t new\_event;
20 time\_type timestamp;
21
22 switch(event) {
23
24 case INIT: // must be ALWAYS implemented
```

```
25 state = (lp\_state\_t *)malloc(sizeof(lp\_state\_t));
26 state->packet\_count = 0;
27 timestamp = (time\_type)(20 * Random());
28 ScheduleNewEvent(me, timestamp, PACKET, NULL, 0);
29 break;
30
31 case PACKET: {
32 pointer->packet\_count++;
33 new\_event\_content.sent\_at = now;
34 int recv = FindReceiver(MESH);
35 timestamp = now + Expent(DELAY);
36 ScheduleNewEvent(recv, timestamp, PACKET, \&new\_event, sizeof(new\_event));
37 }
38 }
39 }
```

The code logic is fairly simple: upon `INIT` event, the LP's state is `malloc`'d and initialized, and an initial packet is sent to the LP itself. Whenever a `PACKET` event is received, a local counter is increased, and a packet is sent back to a random LP in the simulation environment. Timestamps are computed according to an exponential distribution, exploiting the internal `Expent()` function.

`OnGVT()` is the second callback to be implemented, which performs a local check on the LP's state. If the number of packets passed through the LP is smaller than `PACKETS`, then the simulation cannot be halted yet:

```
50 bool OnGVT(lp\_state\_t *snapshot, int gid) {
51 if (snapshot->packet\_count < PACKETS)
52 return false;
53 return true;
54 }
```

Chapter 4

Group of Logical Process

Within a parallel discrete event simulation (PDES), the main element that we take into account is the Logical Process (LP). LP is the basic element of a simulation, it is composed by the state and an event queue, it processes the events, consequently advancing the simulation time (LVT) and changes its state coherently. In detail, each LP is associated with a private clock, which expresses the simulation time up to which it has progressed. The value of this private clock will be referred to as *Local Virtual Time* (LVT), emphasizing the fact that LP_i and LP_j have reached the different (local) simulation time values $LVT_i \neq LVT_j$. At the end of event processing, it can send events to other LPs so as to continue the simulation.

With the increasing spread of shared memories, the simulation platform has to allow the programmers to access, in a completely transparent way, to the states of other LPs, in order to create more complex simulations and to

represent better the reality. This issue is discussed thoroughly [tesi Francesco Nobilia], but through *Event Cross Manager* we can access the memory in a way transparent for the programmer, implementing automatically at platform level and synchronization between LPs. The *Event Cross-State Synchronization* scheme uses different control messages to synchronize the LPs in order to allow a coherent vision of state. The control messages are a particular messages in the system that allow the different LPs to synchronize each other, forcing the LP that receives the `rendezvous-start` to restore a state, consequently the LVT, coherent with the request.

Furthermore, we have also to consider that interactions among processes could happen frequently, and the synchronization cost could reduce performance. In details, we need to synchronize the different LPs otherwise we can access an incoherent state and so we compromise the correctness of entire simulation.

Since we run our simulation in parallel, the resources of a LP could be accessed simultaneously by different LPs on other threads, leading the system to synchronize them and so it does not exploit the parallelism. This behaviour could lead the simulation engine to execute long series of rollback, because, for example, if LP_x wants to read some information inside the LP_y state, but LP_x has simulation time far ahead ($LVT_y < LVT_x$), this triggers a rollback of LP_x. So LP_x is forced back to the time required by LP_y and discards all his later speculative work.

If the two LPs are on different threads this degrades the performance of

the entire simulation, because we have a thread, that handles LP_y, waiting for the other, where is LP_x, to synchronize and breakouts. Once you unlocked the thread where is LP_x, the latter waits for LP_y log into your status and enable him to continue the simulation.
This is very likely to occur frequently between different LPs, this means that the performance of the simulator are considerably degraded and simulation advances very slowly.

As already introduced previously the main objective of this work is to create a new concept of LP: Groups of Logical Processes (GLP). This solution is completely transparent to the programmer, indeed thanks to simulation dynamics we can understand that there exists a relationship between different LPs, and so they need to continue the simulation in a joint fashion in order to increase performance and reduce the likelihood of rollback.
This solution has been implemented and tested in the ROme OpTimistic Simulator (ROOT-Sim)[42].

In this chapter we will make a thorough discussion of what Group of Logical Processes is and the different facets of the group management:

- How we create a group: we examine the conditions that result to group creation, in fact if it is created with the wrong LPs or it is created too early, this could lead to disadvantages in the simulation.

- How we schedule a group: the simulation engine has to determine which LP, inside the group, has to execute first and which event is executed first.

- How to destroy a group: we expose an adaptive solution, that take into account the structure of model, in order to determine the right time at which to destroy a group. In this way we can go back to examining the different interactions between LPs and then determine a new more efficient configuration of the groups.

- How we manage the rollback: of course in the case of groups, even if it is less frequently thanks to this new facility, we must still consider the situation where a set of processes must go back to an earlier time, and then how the simulation engine has to perform the rollback in this situation.

4.1 Creation of Group of Logical Processes

The purpose of group creation is to bring together several LPs into a single group object in order to allow the progress of the simulation of this object in a synchronized manner.

4.1.1 Group Data Structure

The basic object in our simulation is the Logical Process. This is composed of its state and of a queue that contains set of events, which are ordered by timestamp.

The object that we are going to create, called Group of Logical Process

(GLP), is a set of LPs and, according to what said before, a set of states and set of different queues, as shown in Figure 4.1.

The peculiarity of this new object is to allow each LP which belongs to

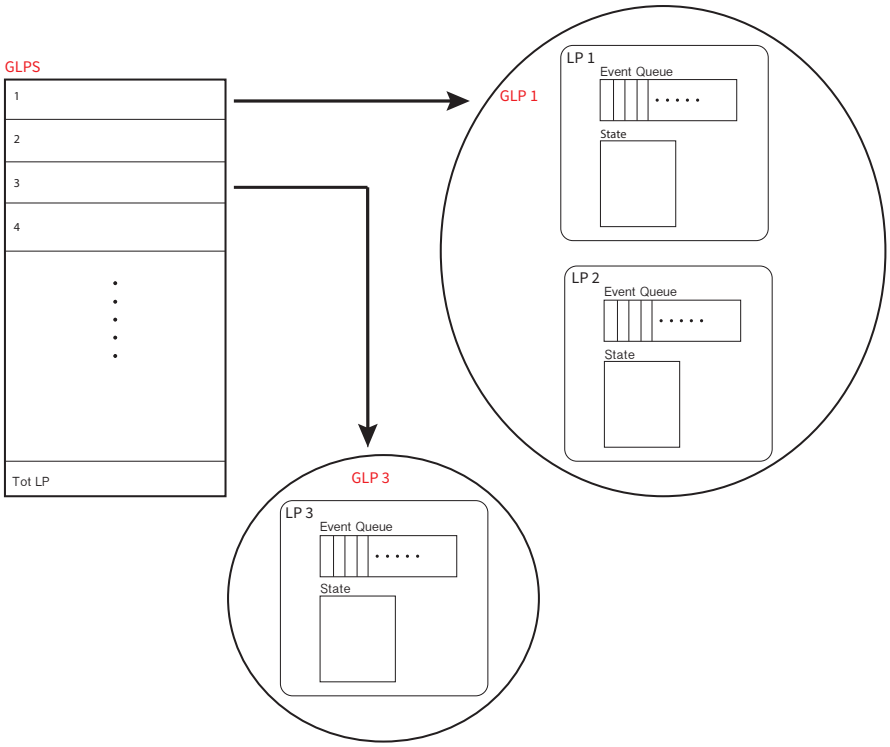

Fig. 4.1: Group of Logical Processes

the group to access independently each state of the other LPs that belong to its group, without incurring in Event Cross State and therefore without forcing synchronization phase with the other. Considering what has been said previously, at simulation startup every Group is composed by only one LP, in particular the one with the same id, then initially we have a number of groups equal to the number of LPs. Going forward, during simulation, some LPs will change group in order to decrease the synchronization cost, we will see this concept in section 4.1.2. If there are not interactions between LPs,

we have a group for each Logical Process, without loss of generality.

To manage groups within our simulator ROOT-Sim [42], we introduce a new global data structure called GLPS, that has a dimension equal to $numLP$, where $numLP$ is the total number of active LPs. Each entry of this data structure is `GLP_state`, which allows us to trace the shape of groups. The `GLP_state` structure is composed by:

- `id`: Identifier of current group.
- `tot_LP`: This field allows us to know the number of LPs contained within a group. At the start-up is set to one, because each group contains exactly one LP.
- `local_LPS`: This field contains a pointer to an array of `LP_state` managed by this group.
- `initial_group_time`: At group start up, this field contains the pointer to the next event with maximum timestamp, if present, otherwise the bound, between all LPs that compose current group. This is useful to understand when the group is ready, and in the rollback phase if we have to consider the classical rollback operation or the group augmented rollback operation.
- `state`: We use this variable to maintain coherently the simulation state of the group according to the simulation states of LPs that it contains. Furthermore, it is useful to block all the other LPs if one of those is in a blocking state.

- `bound`: This is the last correct event executed by the Group, and so the last correct event executed by one of LP inside the Group.
- `from_last_ckpt`: Counter to see how many messages are executed since the last state snapshot
- `ckpt_period`: Every how many messages we have to take a state snapshot
- `counter_rollback`: This field is used during the augmented rollback phase to understand how many LPs already finished the rollback operation and so when the Group can start the silent execution.
- `counter_silent_ex`: Counter to understand when the silent execution phase is done and the group can continue in a `READY` state
- `counter_synch`: At group start up, we have to synch all the LPs at initial group time, so as soon as an LP processes the `START_GROUP` control message, it decrements this counter. When this variable is 0 the group is ready to start.
- `counter_log`: This field is used to create a snapshot of the entire group at the same time. In this way when the group has to roll back, all the LPs have a simultaneous state snapshot.

We have introduced in the data structure `GLP_state` also the field `state`, in order to have a state of the group in line with that of its LPs. We also introduced a new state within the simulation called *Wait_for_Group*. The whole state machine for groups is shown in Figure 4.2.

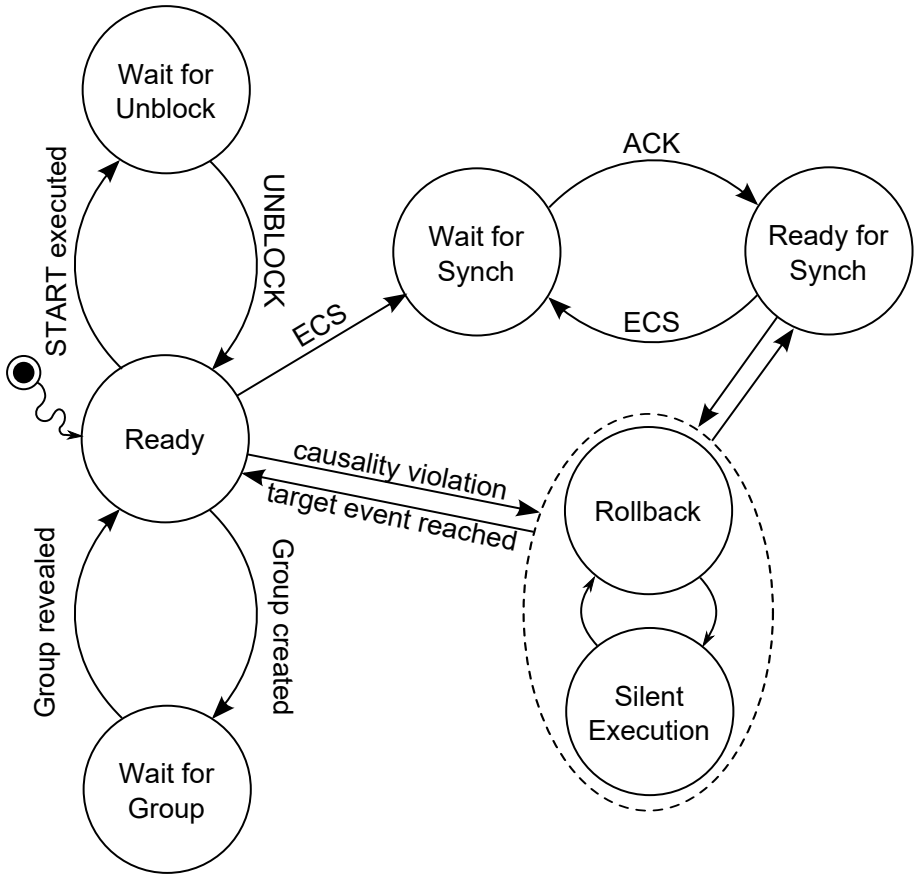

Fig. 4.2: State machine for groups

4.1.2 How and When Groups are Created

In our solution, we consider that the system can transit only from a non-clustered (namely where LPs run independently executing Event Cross State synchronizations) to a state in which the system exploits groups. It is not possible that the simulation transits from one group configuration to another (Figure 4.3). This allows us to ensure consistent execution and avoids the creation of synchronization barriers to make the vision of the group state coherent to every thread.

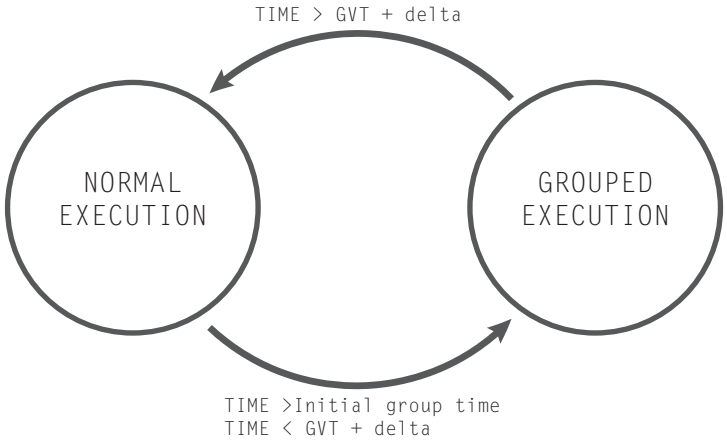

Fig. 4.3: State machine for simulation

As mentioned before, we create a GLP analysing the interaction between different LPs. To keep track of these interactions, we exploit a matrix called *LpDependencies*, with dimension equal to $numLP \times numLP$, where each field represent the number of interactions between two LPs. Indeed, as soon as LP_i performs an access to LP_j's state, the simulation engine detects a cross-state dependency and updates the counter inside $LpDependencies(i, j)$ and $LpDependencies(j, i)$. In this way, every element (i, j) tells how many interactions have taken place between the two LPs.

Therefore, whenever the simulation engine executes an update, it checks if the time at which the current event-cross dependency is performed is far from the last update.
In fact, in addition to storing the number of interactions that have occurred between the LPs, we also store the last timestamp of the event that caused the update. In this way, the system checks whether the timestamp stored within *LpDependencies* plus a threshold is greater than the timestamp of

the current event. If this is true it executes the update, otherwise it resets the counter and performs the update. In other words, if

$$T_{currentECS} - T_{LpDependencies} < t_{freshness}$$

then the simulation engine performs the update. This is done in order to take into account only recent interactions, and discarding the sporadic one, indeed if two LPs interact many time but far away in time, this is an interaction that we have not to taking into account.

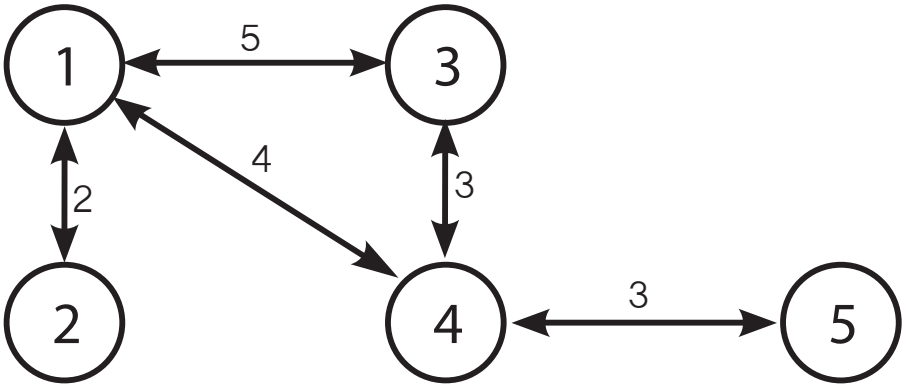

Fig. 4.4: Undirected multigraph to represent LPs interconnection

We can see this matrix as an incidence matrix of an undirected multigraph $G = (V, E)$ where the set of vertices V represents all the LPs inside the simulation and the set of edges E represents, according to what said above, the interconnections between LPs (Figure 4.4).

Upon the GVT computation, one of the worker threads in the system uses the information in the matrix to compute the new groups, depending on the most significant interactions. In detail, for each LP_i the thread determines what is LP_j with which LP_i has performed more interactions.

We compute the new structure of the group only at the GVT because if someone performs a rollback, it can not go back before the GVT. This avoids the situation where two different instances of group overlap. We will explain this in detail in section 4.4.
At the end, the thread computes the index of highest dependency value for each LP as:

$$MaxDep_k = \max_{i \in [0, numLPs-1], i \neq k} \{LpDependencies[k, i]\}.$$

With this information, we can create a new vector, named *LpGroup* vector, with a size equal to $numLP$ and composed by a tuple with the form $\langle MaxDep_k, group \rangle$, but the field group, at the start time, is set to $\emptyset$.

This configuration tells that LP_k, associated with the k-th row of the *LpGroup* vector, has its highest dependency counter set to $MaxDep_k$ and belongs to the special group $\emptyset$, meaning that LP_k still belongs to no group.

This construction transforms the multigraph G into another multigraph $\bar{G}$ such that the set $\bar{V} \equiv V$, but if $\{i, j\} \in \bar{V}$, then $\{i, k\} \notin \bar{V} \quad \forall k \neq j$. This means that every node $i \in \bar{V}$ has at most one edge connecting it to another edge $j \in \bar{V}, i \neq j$, and by construction $j = MaxDep_i$.

We therefore apply a graph visiting algorithm on $\bar{G}$ to determine the groups. We iterate over all indices $k \in [0, numLPs - 1]$, and for each value k we execute the recursive function REGROUP(*LpGroup*, k, $\emptyset$) shown in Algorithm 1. The goal of this recursive function is to determine whether the selected LP already belongs to a group. In the negative case, if the passed

value for the group is not $\emptyset$, then the target LP is aggregated into the passed group (line 6), otherwise a new group is created, associated with the ID of the passed LP (line 8). In the positive case, no action is taken for the current LP, and the group the LP belongs to is returned (line 3). Both cases (namely, lines 6 and 8), are actually *tentative groups*, which could be later confirmed or discarded. In case the LP was associated with a tentative group, a recursive call is issued to REGROUP() (line 11), selecting as the target LP the $MaxDep$ one of the current LP, and passing as the GLP the ID of the group which the current LP belongs to. The GLP of the current LP is then updated with the return value of this call, which is done so as to backwards propagate the creation of new groups or the agglomeration to existing ones (line 13). Line 11 can either confirm a tentative group for a given LP, or supersede it with a different one.

Algorithm 1 Group Construction

```
 1: procedure REGROUP(LpGroup GLP, int LPid, int group)
 2:     if GLP[LPid].group ≠ ∅ then
 3:         return GLP[LPid].group
 4:     end if
 5:     if group ≠ ∅ then
 6:         GLP[LPid].group ← group
 7:     else
 8:         GLP[LPid].group ← LPid
 9:     end if
10:     if GLP[LPid].MaxDep ≠ ∅ then
11:         GLP[LPid].group = REGROUP(GLP, GLP[LPid].MaxDep, GLP[LPid].group)
12:     end if
13:     return GLP[LPid].group
14: end procedure
```

We show in Figure 4.5 an example execution of Algorithm 1 in the case of 8 LPs. In the example, LP_0 exhibits a large number of cross-dependencies towards LP_3, LP_1 shows no cross-state dependencies, LP_2 is dependent on LP_6, LP_3 has no dependencies, LP_4 depends on LP_1, LP_5 depends on LP_6,

LP_6 depends on LP_4. Algorithm 1 is first invoked on LP_0, which belongs to no group (i.e., the *group* field of row 0 of *LpGroup* is set to $\emptyset$), and therefore a new group with ID 0 is created (line 8). Then, since $MaxDep_0 = 3$, line 11 is executed as REGROUP(*LpGroup*, 3, 0). Therefore, for LP_3, the group is set to 0 (line 5), and the value 0 is returned again at line 13, confirming the tentative group. Thus, LP_0 and LP_3 now both belong to group 0. The execution then selects LP_1 which does not belong to any group: the new group 1 is created. Then, LP_2 is selected, which is the most interesting execution case of this example. First, this LP is set to tentative group 2 (line 8), and then the graph visiting selects LP_6. Since LP_6 belongs to no group, the new tentative group 6 is created, and the visiting goes to LP_4. LP_4, similarly to LP_6, creates the new tentative group 4. When the visit reaches LP_1, line 3 is executed, as LP_1 already belongs to group 1. Therefore, all tentative groups for LPs 4, 6, and 2 are backwards superseded by group 1 (as per lines 11 and 13). The actual execution for LPs 3 to 7 can be trivially deduced from the already analysed executions. It is interesting to note that LP_7 belongs to a group composed of a single LP, and it is therefore similar in spirit to the traditional LP-based organization.

Once that new configuration of group is ready, we reset the counter inside the *LpDependencies* and install this new GLP organization, but when the simulation restart the group is not yet revealed.

Indeed, since we execute a speculative simulation, some LPs could be at a virtual time far away, and this could generate consistency errors. For example (Figure 4.6), if the LP_0 accesses the state of LP_1, it sees an incorrect state

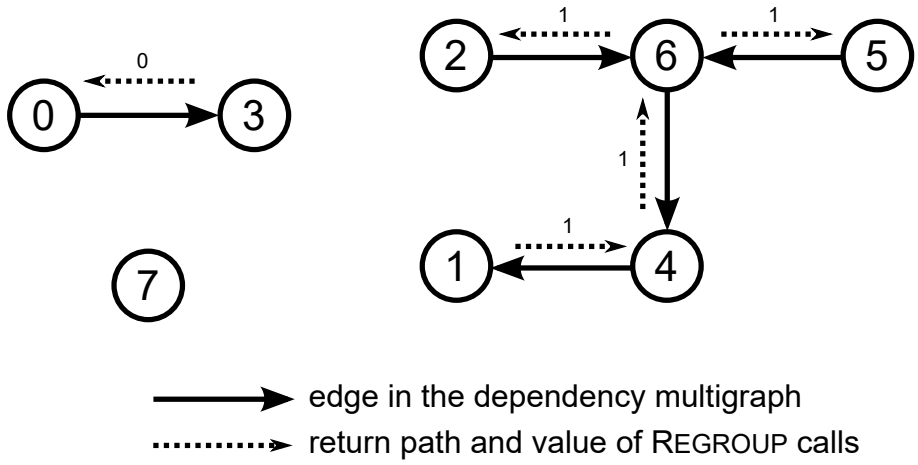

Fig. 4.5: REGROUP execution with 8 LPs.

because LP_1 has LVT bigger than LP_0. According to this new organization, no synchronization message has been sent by LP_0 and then LP_1 does not execute a rollback.

Another problem that could arise is the case in Figure 4.7. In this case LP_1 could start a synchronization phase with another LP, for example LP_2, that does not belong to the group. There is no limit in the amount of wall-clock time to wait for LP_2 to send a rendezvous-ack control message. In the meanwhile, LP_0 can be selected from by the thread for execution and it can continue the simulation. In this way, LP_0 can access the state of LP_1 without sending the rendevous-start and consequently it performs an incorrect execution. In fact, LP_1 has not yet completed the event, because it is waiting for the ack control message from LP_2, and therefore LP_1 accesses an inconsistent state.

To solve this problem, we introduce a new control message, named `START_GROUP`, sent to all the components of new group configuration. The timestamp of

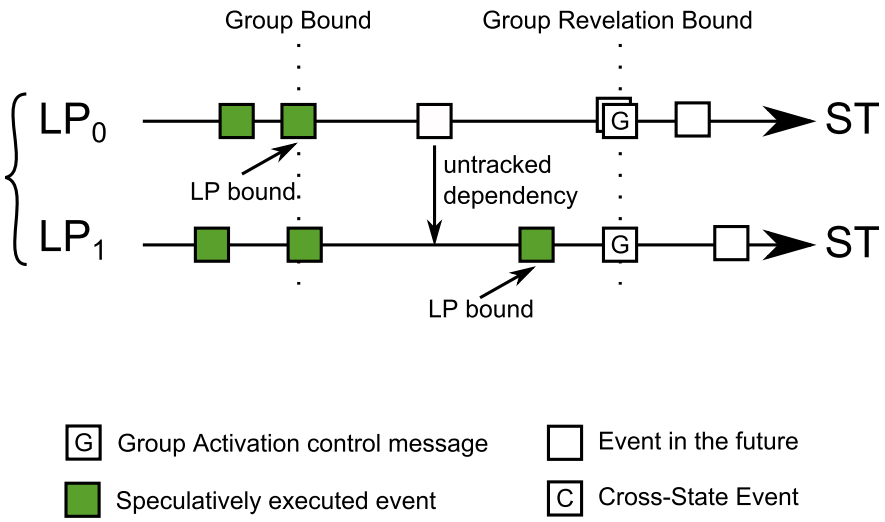

Fig. 4.6: Incoherent memory access without `START_GROUP` control message. Case 1.

`START_GROUP` is computed analysing all the timestamps of next events of all the LPs that belong to the group. If the next event is not present, the timestamp of the bound is consider.

$$T_{start\_group} = \max\{e_k\} \quad \forall k \in \mathcal{G}$$

$$where \quad e_k = next\_event(LP_k) \quad if \quad exist \quad or \quad e_k = bound(LP_k)$$

Therefore, one of these control messages, that the thread that computes the new group structure sent to all the group component is put into the `GLP_state` field `initial_group_time`, the `counter_synch` is set to `tot_LP` and the state of GLP is set to `GLP_STATE_WAIT_FOR_GROUP`.

As soon as an LP processes a `START_GROUP` message, it decrements the `counter_synch` and sets its state to `LP_STATE_WAIT_FOR_GROUP`. This is a

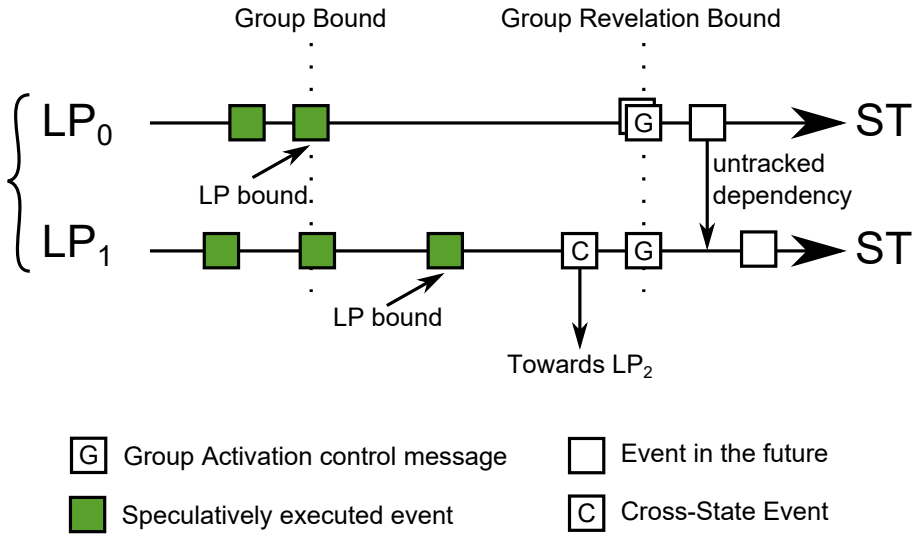

Fig. 4.7: Incoherent memory access without `START_GROUP` control message. Case 2.

blocking state, in this way all the LPs have to wait for the other components. As soon as all have executed `START_GROUP` message, the counter is zero and the group is set to `GLP_STATE_READY`, in this way the new group execution can start.

4.2 Schedule of Group of Logical Processes

From the moment that the idea behind the execution of an LP changes, we have also to change the scheduling logic of Group of Logical Processes.
The main change that we introduce is how we organize the LPs over the different threads available. In particular, we have to assign all the LPs of a Group to a single thread. This restriction is because otherwise it could happen that an LP over a different thread executes a lot of events, modifies

its state, and increases its LVT. If one LP, bound to another thread but with the same group, accesses the state, it is not consistent.
Since the next event scheduled by the thread is the smallest timestamp among all the event queues available, we can ensure that the Group executes all the events in timestamp order, and consequently all the events see the correct state of the other LPs.
By adding only the rule on how the LPs are distributed on the various threads, we have not to change the logic of events processing, because as mentioned above, this guarantee us a successful execution.

Since the idea behind selection of the next LP does not change, we have to align the group state with the LP state. Indeed, if all the LPs have processed the `START_GROUP` control message, each time that one LP inside the group updates its bound, it updates as well the group bound, in this way it is the last correct event processed by one of the LPs in the group. Therefore if one of those LPs executes a rendezvous-start control message, the group has to set the state to `GLP_STATE_WAIT_FOR_SYNCH` in order to avoid that other LPs continue the execution. Indeed, also if the LP is in a blocked state, the other LPs could continue the simulation. This case is also for the rendezvous-ack control message, and in order to keep the state consistent we set the group state to `GLP_STATE_WAIT_FOR_UNBLOCK`.

The overall algorithm to compute and install groups is reported in Algorithm 2. For the sake of performance, it is devised mostly as a non-blocking algorithm [43], except for a final synchronization point where all the worker threads hit a barrier. The idea behind this wait-free algorithm is that while

one thread (which we refer to as the *master thread* of the system, as the check at line 6) computes the new groups, it is pointless for the other worker threads to wait for this task to complete. In fact, this would waste a significant amount of CPU time which could be used to execute simulation events. To let other worker threads continue processing, we rely on counters which described the *group determination era*, namely a global shared counter (*group_era*) and a set of thread-private counters (*my_group_era*). When the master thread starts computing the new GLPs, the new group control blocks (which keep as well pointers to LP-related information) are computed on a global data structure. When the GLPs are all determined, via repeated calls to REGROUP(), the global *group_era* counter is incremented (line 10). At every main simulation loop cycle, all the other worker threads compare the value of *group_era* with their private value of *my_group_era* (line 16), and only if it is incremented they start installing the groups (line 20, for other worker threads). In this way, even though the GLP determination procedure takes a bit of time, the speculative execution can continue at other worker threads. Installing groups requires all the worker threads to make a private copy of the shared group control blocks in thread-private storage, and rebinding the LPs belonging to their groups on them. This wait-free algorithm requires an additional modification, related to the determination of $\hat{e}$. In fact, this should be done by each worker only after that the GLPs have been bound to them (lines 14 and 22). The thread barrier at the end of the algorithm is required so as to ensure that when the new GLPs are determined, no worker thread restarts executing events before all the worker threads have a coherent view on what LPs they are in charge of scheduling.

Algorithm 2 Group Determination and Installation

```
 1: new_LP_groups[]
 2: LP_groups[] (thread-private)
 3: group_era
 4: my_group_era (thread-private)
 5: procedure GroupCompute( )
 6:     if MasterThread( ) then
 7:         for i ∈ [0, numLPs − 1] do
 8:             Regroup(LpGroup, i, ⊥)
 9:         end for
10:         group_era ← group_era + 1
11:         SanitizeGroups( )
12:         InstallGroup( )
13:         ThreadBarrier( )
14:         Compute Group Revelation Bounds for all bound groups
15:     else
16:         if my_group_era == group_era then
17:             return
18:         end if
19:         my_group_era ← group_era
20:         InstallGroup( )
21:         ThreadBarrier( )
22:         Compute Group Revelation Bounds for all bound groups
23:     end if
24: end procedure
```

An additional operation is required, related to the symmetric multi-threaded organization. In particular, it could be the case that due to the detected cross-state dependencies the number of determined groups is smaller that the total number of available cores. Since the architectural organization requires to have a worker thread per available core, in this situation some worker thread might not have any LP to schedule. This has a twofold negative effect: i) the CPU time of these unloaded worker threads is wasted; ii) since all the worker threads cooperate to reduce the GVT value, it would not advance, thus preventing fossil collection and termination detection. This negative effect cannot be neglected, so the master threads executes the SanitizeGroups() routine (line 11), which takes care of this. In particular, in case the number of GLPs $|\mathcal{G}|$ is smaller than the available number of cores C, additional $C - |\mathcal{G}|$ groups are instantiated. To determine which LPs should

fall into these new groups, the *LpDependencies* matrix is scanned, so as to determine what are the LPs with the smallest number of interactions, and they are taken out of their determined groups according to a greedy approach.

4.3 Destruction of Group of Logical Processes

This configuration is valid until that we compute a new GVT_{new} with a LVT bigger than $GTV_{old} + \Delta$. We compute this barrier exploiting the statistics information about the model during the execution, in this way the Δ value is adapts to the execution speed of the model.

At the start up of group, in addition to the `START_GROUP` control message, the worker thread also sends a close message to the group, called `CLOSE_GROUP`. The timestamp associated with this message is exactly $GVT_{old}+$ Δ. In this way, as in the case of initialization of the group, we ensure that all LPs brought out of the group at the same time. Moreover, in the moment in which the LPs process this message, take a snapshot of their current status. This ensures that if one of the LP will perform a rollback and if the timestamp of straggler message will be beyond the end of the group, then we should not bring all LPs to a situation in which the group was revealed only because one of them may not have a status snapshot.

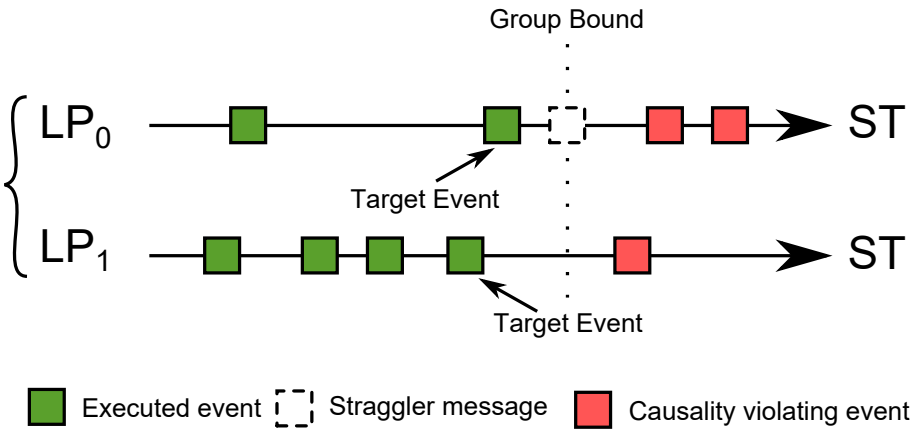

Fig. 4.8: Group Rollback

4.4 Rollback of Group of Logical Processes

Concerning the reception of straggler message, the execution of rollback operation must take into account GLPs, rather than single LPs. In fact, a straggler must rollback the entire GLP, as show in Figure 4.8.

If we do not consider a new management of the rollback, the situation that can occur is when an LP_0 rolls back due to a straggle message, it performs phase costing forward and finally executes the straggler message.

If the LP_0 message needs to access the status of the other, but their bound has a timestamp greater than the straggler, LP_0 could access not consistent information.

To solve this problem you need to bring all the LPs of the group at a time less than or equal to the straggler timestamp to provide a uniform state vision. To perform this operation we select inside each $LP_i \in \mathcal{G}$ a message e_i with timestamp lesser or equal to the $T_{straggler}$. This message represents the last

message that the LP_i has to perform in costing forward.

The goal of the rollback operation is thus to realign the LP bound of every $LP_i, i \in [0, |\mathcal{G} - 1|]$ to $\bar{e}_i$. In case the simulation engine bases its rollback operation upon checkpoint/restore primitives, it would appear sufficient to select at each LP_i the simulation state checkpoint S_i^t such that $t < T_{str}$, and then execute the coasting forward phase. Unfortunately, this simple solution could be easily proven wrong in a twofold way. Consider, in fact, the example shown in Figure 4.9, where GLP $\mathcal{G}$ is already revealed. In this example there

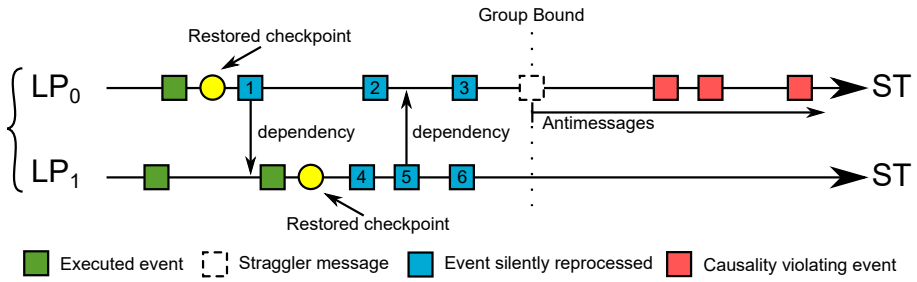

Fig. 4.9: Inconsistent execution of traditional rollback/coasting forward phase with active groups.

are some execution errors. First of all, the LP_0 restore a state before the LP_1 state, and then executes a message (e_1) in costing forward that accesses to the LP_1 state, but the latter is ahead in time. Another error is that if we perform costing forward execution according to the traditional [10], the LP_0 performs costing forward by running all its events in a single moment, and reporting their status to the event immediately before the straggler $e_{straggler}$. When LP_1 executes its costing forward he could access the status of LP_0, and then runs an event-cross dependences without being traced.

To solve these problems we need to modify the execution of costing for-

ward, so as to be consistent with the execution group, and we have to introduce new control message.

We have redesigned the costing forward in order to ensure that also in this case the execution is consistent. In fact, first of all the involved LPs restore its status, decrement the `counter_rollback`. As soon as the counter is equal to zero, it means that all the LPs are ready to execute the coasting forward phase and then we change the group state in `GLP_STATE_SILENT_EXECUTION`. Now, all the events involved in the rollback are scheduled exactly as they were performed initially, referring to the example can expect executed in order e_1 e_2 e_4 e_5 e_6 e_3, so as to provide all involved LPs always a consistent state.

The control message which we have added, called `LOG_GROUP`, guarantees that every $LP_i \in \mathcal{G}$ always has a state snapshot at the same time of the other, in order to avoid cascading rollback and solve the first problem shown. In fact, as soon as an LP of the group decides to take a snapshot, sends a `LOG_GROUP` message with the same timestamp of its snapshot to all GLP (Figure 4.10). In this way, as soon as the group must perform a rollback, there is at least a snapshot common to all LPs. Consequentially, all the data structure inside the `LP_status` to take periodically checkpoint are not more consider, but in group scheduling we take into account the same statistic for group (`from_last_ckpt` & `ckpt_period`).

In case a rollback operation falls before a GLP is revealed, which is detected by checking if the timestamp of a straggler message falls before the group revelation bound, then the group state is brought back to the

`WAIT_FOR_GROUP` state, consequently the cross-state dependencies are again detected by the ECS manager.

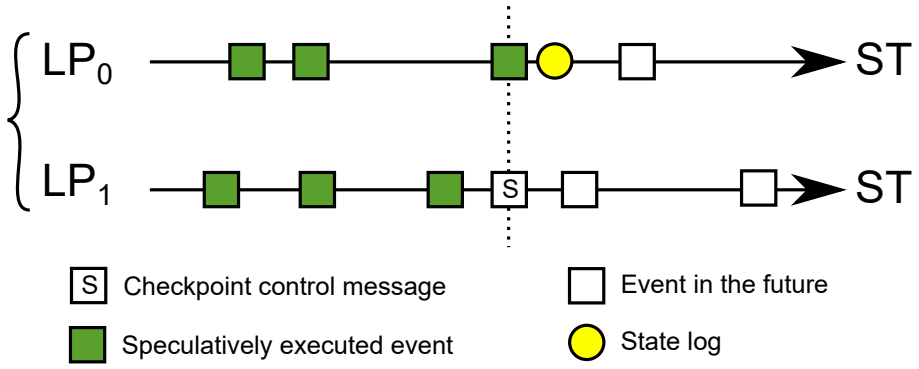

Fig. 4.10: Group checkpoint control message.

Chapter 5

Experimental Evaluation

In this section we provide experimental data achieved by testing our proposal running the implementation of a multi-robot exploration and mapping simulation model, as developed in [44] according to the results in [45].
In this model, 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 un-

explored 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. 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 rendezvous point (not to be confused with rendezvous control messages in the underlying Time Warp platform supporting granulation) 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. 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 [45]) 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.

Discovering the presence of a nearby robot is a crucial step while coding this simulation model. In fact, in case of reliance on classical PDES programming schemes not based on cross-state access, either the robots must communicate to each other their current position (thus exponentially increasing the number of exchanged messages, say cross-scheduled events, 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.
Additionally, estimating 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 [46]. Third, exchanging map information could entail a data transfer non-negligible in size, posing a huge burden on the communication subsystem.

This model is therefore a good test-case for exploiting the innovative programming paradigm based on cross-state access, and to test the advantages from granulating LPs according to the new mechanisms we have presented (just supporting this programming model). In our implementation (as said aligned with the one in [44]), we rely on two different types of LPs, 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 modelling 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 LP instantiates in its private simulation state (by relying on a traditional `malloc` call) a *presence vector*. Each entry of the vector is associated with a specific robot. Whenever a robot enters a given cell, it explicitly informs the LP taking care of the cell's state by exchanging an event, piggybacking a pointer to a buffer in the robot's fsimulation state which keeps the representation of the explored map.
When the cell processes this event, it stores the pointer in the presence vector, which is then scanned to synchronize the information in the map. In particular, all the robots' states are in-place accessed, so as to copy the information from one state to the other. This operation clearly triggers cross-state synchronization and may lead to granulate LPs temporarily residing in a given area, together with the LP modelling the specific portion of the environment where they reside.

To test the GLP proposal we have compared the execution time for this

simulation model when run with the granulation support active, and without granulation thus running with the baseline cross-state synchronization protocol (labelled as CS). We have also run the same identical model on top of a serial engine based on a classical calendar-queue scheduler. Finally, for completeness of the analysis, we have run a version of the same model coded by only relying on the traditional paradigm where cross-state access is not employed/supported, thus basing the interactions among the different parts/entities in the model exclusively on the cross-scheduling of events across the different LPs. For all the tests we run a model with 1000 LPs, the 10% of which represent robots, and the remaining 90% represent sub-regions of the overall bi-dimensional region to be explored.

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 64 GB 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. For the parallel runs we configured the simulation platform to use 32 worker threads.

The total execution time for the simulations are reported in Figure 5.1 for the different settings of the underlying simulation engine (where each reported sample is averaged over 10 runs). For GLP-based runs, we have also considered the variation of the threshold parameter t_{dep}, which we recall can be used to filter out cross-state dependencies that are less valuable (say, their volume is lover than others) while building the GLPs. Also, in GLP-based runs, the granulation process is actuated after the first GVT computation

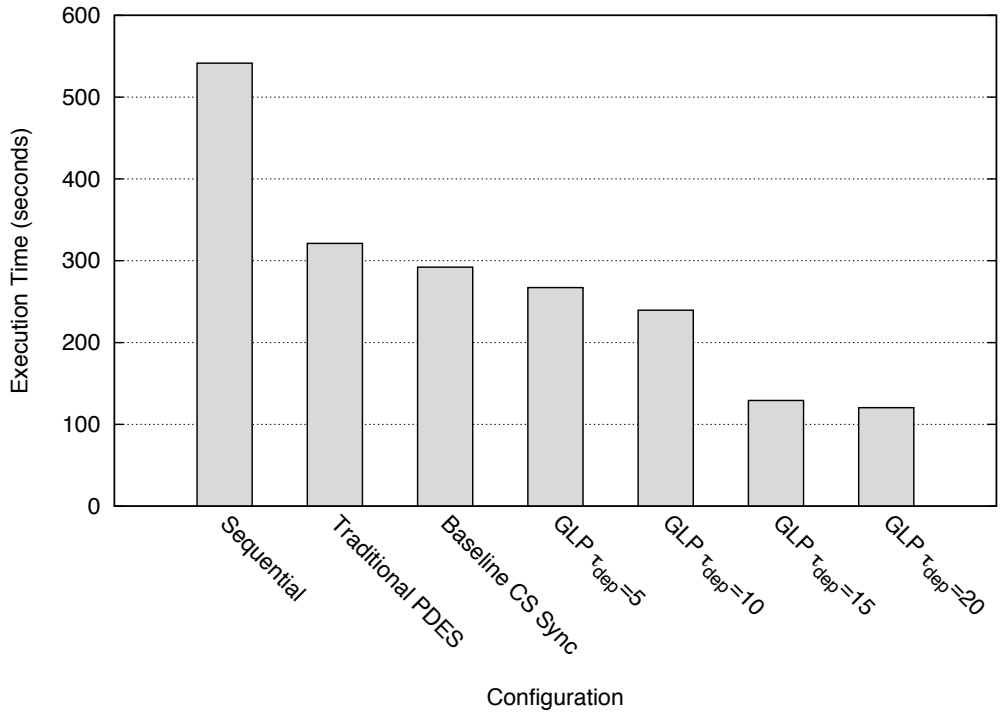

Fig. 5.1: Total Execution Time

round that is subsequent to the fading (due to lifetime expiration) of previously formed groups[1].

By the results we observe that all the platform configurations offering support for in-place cross-state access outperform the traditional PDES scenario, not admitting cross-state accesses. Nevertheless, the baseline CS synchronization scheme offers a limited gain, while the integration of the new granulation support leads to noticeable performance improvements. These improvements are better for larger values of the parameter t_{dep}, indicating that a more accurate selection of actually valuable cross-state dependencies can drive significantly better granulation decisions, leading to more efficient

[1]In all the parallel runs the GVT period has been set to 1 second.

synchronization dynamics, which lead the execution time to be almost 3 times lower compared to both the traditional PDES case and the baseline CS synchronization scheme. An additional observation concerns the step reduction of the execution time with GLPs when increasing t_{dep} from 10 to 15. From the collected statistics we noted that this phenomenon is due to the fact that up to the value $t_{dep} = 10$, scenarios were generated where granulation involved multiple LPs representing distinct subregions, which dos not favor concurrency since the model let each robot LP to move around, with the constraint of residing at any time in a single region. Hence, at any time instant, some event can only lead to cross-state access involving a region and its hosted robots. Different subregions, and their currently hosted robots, should be therefore allowed to execute concurrently to favor parallelism, by placing them in different GLPs.

Chapter 6

Conclusion

In this thesis we have introduced the concept of Group of Logical Processes, and the design of a run-time PDES platform enabling the granulation process. Indeed we augmented the event cross-state protocol in order to understand the synergic interconnection between different LPs.

As soon as we determinate a Group of Logical Process, each objects accesses to the state of the other without any synchronization protocol and under the assumption that any causality violation may rise between the LPs of the group. Indeed, this is obtained binding the entire group, consequently each LP, over only one worker thread. Furthermore, we exposed a programming model where it is not necessary exchange the information using messages, but we can send a pointer inside the message and, according to event cross-state protocol, access to the variable inside the state of another LP. This facilities are completely transparent to the programmer, that coding in a sequential

style.

Such cross-state accesses are what drive the formation of group objects, which are aimed at clustering the baseline objects that along specific execution phases shown larger volumes of cross-state dependencies. Also, this mechanism leads to run-time configurations where the level of parallelism is dynamically determined on the basis of the level of coupling of objects, as determined by cross-state dependencies materialization. We tested our proposal against traditional Time Warp and a variant with cross-state support but without grouping the LPs, for the case of a multi-robot exploration simulation model run on a 32-core machine. By the study we report a 3x improvement in the model execution speed thanks to our proposal.

Bibliography

[1] Frederick Wieland, Lawrence Hawley, A Feinberg, M DiLorento, L Blume, P Reiher, B Beckman, P Hontalas, S Bellenot, and DR Jefferson. Distributed combat simulation and time warp: The model and its performance. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 14–20. PO Box, 1989.

[2] Divyakant Agrawal and Jonathan R. Agre. Replicated objects in time warp simulations. In *Proceedings of the 24th Conference on Winter Simulation*, WSC '92, pages 657–664, New York, NY, USA, 1992. ACM.

[3] Philip Hontalas, Brian Beckman, M DiLorento, Leo Blume, Peter Reiher, Kathy Sturdevant, LV Warren, John Wedel, Fred Wieland, and David Jefferson. Performance of the colliding pucks simulation on the time warp operating system. *Distributed Simulation*, 21(2):3–7, 1989.

[4] K Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, (5):440–452, 1979.

[5] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.

[6] R. M. Fujimoto. The virtual time machine. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 199–208, New York, NY, USA, 1989. ACM.

[7] Paul F Reynolds Jr. A spectrum of options for parallel simulation. In *Proceedings of the 20th conference on Winter simulation*, pages 325–332. ACM, 1988.

[8] David M Nicol. Principles of conservative parallel simulation. In *Proceedings of the 28th conference on Winter simulation*, pages 128–135. IEEE Computer Society, 1996.

[9] Richard Fujimoto and David Nicol. State of the art in parallel simulation. In *Proceedings of the 24th conference on Winter simulation*, pages 246–254. ACM, 1992.

[10] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[11] David Jefferson, Brian Beckman, Frederick Wieland, Leo Blume, and Mike DiLoreto. *Time warp operating system*, volume 21. ACM, 1987.

[12] Li-li Chen, Ya-shuai Lu, Yi-ping Yao, Shao-liang Peng, and Ling-da Wu. A well-balanced time warp system on multi-core environments. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and*

Distributed Simulation, PADS '11, pages 1–9, Washington, DC, USA, 2011. IEEE Computer Society.

[13] Elizabeth Whitaker Lynch and George F Riley. Hardware supported time synchronization in multi-core architectures. In *Principles of Advanced and Distributed Simulation, 2009. PADS'09. ACM/IEEE/SCS 23rd Workshop on*, pages 88–94. IEEE, 2009.

[14] Elizabeth Whitaker Lynch and George F Riley. A sensitivity analysis of a new hardware-supported global synchronization unit. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–4. IEEE, 2009.

[15] Jun Wang and Carl Tropper. Selecting gvt interval for time-warp-based distributed simulation using reinforcement learning technique. In *Proceedings of the 2009 Spring Simulation Multiconference*, page 49. Society for Computer Simulation International, 2009.

[16] Jeffrey S Steinman, Craig A Lee, Linda F Wilson, and David M Nicol. Global virtual time and distributed synchronization. In *ACM SIGSIM Simulation Digest*, volume 25, pages 139–148. IEEE Computer Society, 1995.

[17] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[18] Steven Bellenot. State skipping performance with the time warp operating system. In *6th Workshop on Parallel and Distributed Simulation*, volume 24, pages 53–64, 1992.

[19] Yi-Bing Lin and Ed D Lazowska. Reducing the saving overhead for time warp parallel simulation. *University of Washington Department of Computer Science and Engineering*, page 79, 1990.

[20] Josef Fleischmann and Philip A. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. *SIGSIM Simul. Dig.*, 25(1):50–58, July 1995.

[21] Avinash C Palaniswamy and Philip A Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *ACM SIGSIM Simulation Digest*, volume 23, pages 127–134. ACM, 1993.

[22] Herbert Bauer and Christian Sporrer. Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving. In *Simulation Symposium, 1993. Proceedings., 26th Annual*, pages 12–20. IEEE, 1993.

[23] Robert Rönngren, Michael Liljenstam, Rassul Ayani, and Johan Montagnat. Transparent incremental state saving in time warp parallel discrete event simulation. In *ACM SIGSIM Simulation Digest*, volume 26, pages 70–77. IEEE Computer Society, 1996.

[24] Darrin West and Kiran Panesar. Automatic incremental state saving. In *ACM SIGSIM Simulation Digest*, volume 26, pages 78–85. IEEE Computer Society, 1996.

[25] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Didymelor: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53. IEEE Computer Society, 2009.

[26] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 319–327. IEEE, 2010.

[27] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 9(3):224–253, 1999.

[28] Justin M LaPre, Elsa J Gonsiorowski, and Christopher D Carothers. Lorain: a step closer to the pdes'holy grail'. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation*, pages 3–14. ACM, 2014.

[29] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[30] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. Gtw: a time warp system for shared memory multiprocessors. In *Simulation Conference Proceedings, 1994. Winter*, pages 1332–1339. IEEE, 1994.

[31] Alessandro Fabbri and Lorenzo Donatiello. Sqtw: A mechanism for state-dependent parallel simulation. description and experimental study. In *Proceedings of the Eleventh Workshop on Parallel and Distributed Simulation*, PADS '97, pages 82–89, Washington, DC, USA, 1997. IEEE Computer Society.

[32] Michael Lees, Brian Logan, Rob Minson, Ton Oguara, and Georgios Theodoropoulos. Distributed simulation of mas. In *Multi-Agent and Multi-Agent-Based Simulation*, pages 25–36. Springer, 2005.

[33] B. Logan and G. Theodoropoulos. The distributed simulation of multi-agent systems. *Proceedings of the IEEE*, 89(2):174–185, Feb 2001.

[34] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 69–80. ACM, 2007.

[35] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[36] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 2008.

[37] Michael F Spear, Virendra J Marathe, William N Scherer III, and Michael L Scott. Conflict detection and validation strategies for software transactional memory. In *Distributed Computing*, pages 179–193. Springer, 2006.

[38] Horst Mehl and Stefan Hammes. How to integrate shared variables in distributed simulation. *SIGSIM Simul. Dig.*, 25(2):14–41, September 1995.

[39] MPI Forum. Message Passing Interface Forum. http://www.mpi-forum.org/, 1994.

[40] Roberto Toccaceli and Francesco Quaglia. DyMeLoR: Dynamic Memory Logger and Restorer Library for Optimistic Simulation Objects with Generic Memory Layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172. IEEE Computer Society, 2008.

[41] Francesco Antonacci, Alessandro Pellegrini, and Francesco Quaglia. Consistent and efficient output-streams management in optimistic simulation platforms. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pages 315–326. ACM, 2013.

[42] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. The rome optimistic simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 96–98, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[43] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[44] Alessandro Pellegrini and Francesco Quaglia. Programmability and Performance of Parallel ECS-based Simulation of Multi-Agent Exploration Models. In *Proceedings of the 2nd Workshop on Parallel and Distributed Agent-Based Simulations*, Porto, Portugal, 2014. LNCS, Springer-Verlag.

[45] Dieter Fox, Jonathan Ko, Kurt Konolige, Benson Limketkai, Dirk Schulz, and Benjamin Stewart. Distributed Multirobot Exploration and Mapping. *Proceedings of the IEEE*, 94(7):1325–1339, 2006.

[46] H Mehl. A deterministic tie-breaking scheme for sequential and distributed simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation*. ACM, 1992.

Acknowledgements

First of all, I wish to ...