SAPIENZA
UNIVERSITÀ DI ROMA

FACULTY OF INFORMATION ENGINEERING,
INFORMATICS AND STATISTICS

Master's Degree Thesis in

ENGINEERING IN COMPUTER SCIENCE

Transparent Distributed Cross-State Synchronization in Optimistic Parallel Discrete Event Simulation

Advisor
Prof. Bruno Ciciani

Candidate
Matteo Principe

Co-Advisor
Dr. Alessandro Pellegrini

External Advisor
Prof. Francesco Quaglia

Academic Year 2016/2017

Si sopravvive di ciò che si riceve
si vive di ciò che si dona

Contents

List of Algorithms

List of Figures

Abstract

Parallel Discrete Event Simulation (PDES) is a powerful technique to simulate real world complex models. In fact, by sharing the workload over different machines, thus parallelizing it over different entities it is possible to satisfy the high amount of computational power and resources needed by such a kind of models. Indeed, taking advantage of this sort of organization allows to overcome both the *power wall* ([Sut05]) and the *memory wall* ([McK04]), which represent the main aspects limiting the delivery of high performance executions.

In particular, this thesis directly faces the disadvantages given by distributed memory accesses arising between Logical Processes (LPs, which are the main simulation entities representing real world objects evolving over time, [Fuj90]) while synchronizing between each other. In fact, those LPs are continuously communicating and often requesting access to portions of memory owned by others. This scenario needs the involved (two or more) LPs to synchronize in order to ensure that the requested operation will be correctly reflected in their memory. This could lead of course to a degradation of performance, given the distributed nature of the system on top of which the simulations are executed. The innovative technique presented in this document exploits the new kernel-level facilities in order to detect, manage and optimize the aforementioned situation, which is known as *Event Cross State Synchronization* (ECS).

Also, an important result reached in this work is represented by the fact that the provided solution is transparent to the final model developer in the sense that the whole job is handled by the underlying run-time environment, leveraging shared-memory accesses of the simulation state by relying on the data-sharing paradigm and executing and deploying in a seamless manner on distributed memory within clusters of multicore machines.

The whole work was developed on top of an open source, optimistic simulation platform provided by the High Performance and Dependable Computing Systems (HPDCS[1]) research group at Sapienza, University of Rome, called **RO**me **O**p**T**imistic **Sim**ulator (ROOT-Sim[2]).

[1] https://hpdcs.github.io/.
[2] https://github.com/HPDCS/ROOT-Sim.

The remainder of this thesis is organized as follows. In the first chapter, an overview and an explanation of the main topics of simulation and PDES is presented, focusing in particular on the environment and the choices made in this work. In Chapter 2, an introduction to the programming models of PDES is given, going through what literature proposes. Then, in Chapter 3, the proposed solution is discussed, concentrating on the specific implementation and design choices that were made in order to get the best results. Finally, Chapter 4 shows the experimental data for an assessment of the proposed solution.

Chapter 1

Introduction to Simulation and PDES Overview

The term *simulation* comes from the Latin expression *simulare*, which means the act of imitating a real world situation evolving over time. In order to accomplish this, it is needed to build an abstraction of the physical system of interest, trying to make it follow the specific behavior of the real one: in other words, a model is needed. Note, however, that this is not an easy task, since sometimes it can be the case that our knowledge about the reality around us is not enough to produce a model, or that the generated model is too complex to be simulated, and this is why research, is giving effort in this area. Historically, research about simulation started in late 1979 by Chandy and Misra in [CM79].

In particular, two main problems brought scientists and researchers to discriminate different kinds of simulation: time and randomization. The first one is related to how the passing of time is considered, since it can be seen as continuous, which is well suited for real world phenomena such as heat diffusion, or discrete, that fits for example the case of the demographic growth of a specific area, where the time interval between evolutions of the model could be enlarged (e.g.: a day, a month...).

As far as randomization is concerned, it can be noticed that not all the simulation applications are strictly related to timing. In fact, there are some cases where it is not an interesting quantity, for example when the target is to evaluate or approximate a mathematical problem which is not easy to solve analytically and which is involving a high number of variables. Consider, for instance, the case of the approximation of the value of finite integral of a function. It is fundamental to pick as more randomly distributed points falling in the integration interval as possible, in order to sum all the values the function gets in those points and dividing them by the total number of points.

These different approaches gave rise to three main different types of simulation:

- Continuous simulation
- Monte Carlo simulation
- Discrete event simulation

1.1 Simulation Taxonomy

Depending on what kind of phenomenon the simulation needs to carry on, a specific kind of simulation can be suited or not. Of course, this means also that there isn't a type, among the aforementioned ones, that outperforms another since the best is to be chosen with respect to the kind of simulation of interest. However, in this work we focus on the particular case of Discrete Event Simulation. In Figure 1.1 a scheme of the main simulation types is shown.

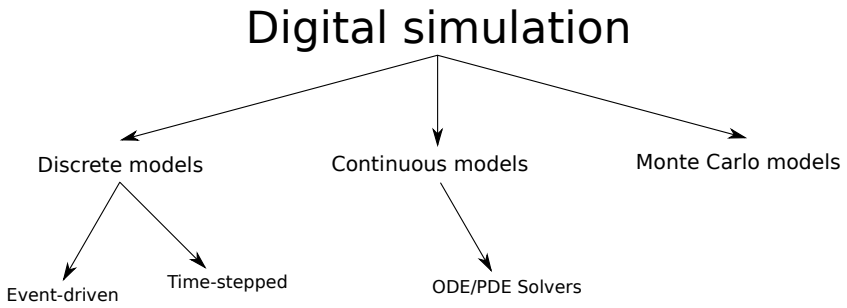

Figure 1.1: Simulation taxonomy diagram

1.1.1 Discrete Event Simulation

Discrete Event Simulation (DES) is based on the representation of the simulated system's evolution as a chronological sequence of events. An event is said to be discrete if it is related to a specific instant of time, and if it marks a change in the simulation's state in a infinitesimal period of time (namely, impulsively). If this type of simulation is run on top of a parallel (possibly distributed) system, it is called *Parallel Discrete Event Simulation* (PDES).

Events are generated and processed by simulation entities called *Logical Processes* (LPs), each one of them representing a real-world object evolving in time due to state transitions, which are dictated by the actual model. Thus, we can define a simulation as a collection of N logical processes, $LP_0, LP_1, \ldots, LP_{N-1}$,

each of them keeping a state S_j , $j \in [0, N-1]$) containing all the data used to *i*) execute, asynchronously or not, events in an independent manner with respect to other LPs, *ii*) produce events targeting either itself or other LPs, *iii*) interact with other LPs through message exchange. It is important to notice that every logical process executes events according to its *Local Virtual Time* (LVT) which is a private clock expressing the simulation time instant the specific LP has reached and which is different from the simulation time, representing the actual amount of time taken to perform the simulation, called *Wall Clock Time* (WCT).

When dealing with DES, a set of building blocks must be specified:

- **Clock** Since logical processes execute independently, the probability that their LVT diverges from the ones of other LPs is very high. Nevertheless, a global clock shared among all entities of a simulation is an essential element when coordination is needed. Thus, simulation must compute a *Global Virtual Time* (GVT), keeping in mind that, being discrete, time just jumps to next event's timestamp.

- **Random-Number Generators** Being the real-world systems highly dependable on random happenings, a simulation needs to try to recreate such an unpredictable operation. This is why a random number generator is used, which typically is implemented in the form of a *Pseudorandom-Number Generator* (PRNG), since a simulation could need to rerun with exactly the same previously generated values (thus, acting precisely in the same way).

- **Statistics** When running a simulation, collecting interesting data is quite important in order to realize what the model is telling us. Thus, a simulation usually monitors the aspects of interest, in order to allow the developers to tune it and get better performance, or have a better knowledge of what is going on.

- **Event Lists** By definition, this kind of simulation must keep at least an event list (namely, the *pending event* list). The main reason behind the needed presence of this list lies in the fact that the fanout of an event (i.e., the number of events generated during the execution of a single event) can be higher than 1. We need then a method to store events, ensuring a future execution of those events. Usually, more than a list is maintained, since events can be discriminated according to their type and typically are sorted by events' timestamp, guaranteeing that when extracting entries, the processing order is chronological.

- **Ending Condition** Since, once started, a simulation can run indefinitely, the model designer must specify a terminating condition at which

the run should end. For instance, the condition could be the reaching of a specified instant of time or, more generally, of a specific number of processed events.

Discrete Event Simulation is thus strongly related to a particular kind of programming, called *Event-Driven Programming*. This programming paradigm is based on two main phases:

- Event selection/detection
- Event handling

The first one consists in capturing simulation events resembling hardware system behaviors upon interrupts arrival, while the latter deals with the actual execution of actions specified by the event, consequently modifying the simulation state. Of course, event handling could be improved by event dispatching, that manages the association of an event to the corresponding handler, discriminating events according to their type. In order to implement those two phases, typically DES provides a main simulation loop, where essentially *i*) the next event (e.g.: the smallest timestamped one, cf. [Fuj90]) is taken into account (event selection), *ii*) the logic to execute the selected event's actions is triggered (event handling), along with an update of the simulation time with the value of the timestamp of that event, reflecting the actual advancement. However, this is not enough: an initialization phase is needed since the simulation requires the ending condition, the initial simulation state and the initial simulation clock time to be set to some predefined values and also the first event (namely, the *init* event) starting all the other ones of the whole simulation, needs to be processed. In Algorithm 1 the reader can see an example of a high-level implementation of the paradigm discussed above.

Algorithm 1 DES Skeleton

```
1:  procedure INIT
2:      End ← false
3:      initialize State, Clock
4:      schedule INIT
5:  end procedure
6:  procedure SIMULATION-LOOP
7:      while End ==false do
8:          Clock ← next event's time
9:          Process next event
10:         Update statistics
11:     end while
12: end procedure
```

1.2 Synchronization Approaches

As mentioned, simulation models can be arbitrarily complex, making their execution on a single processing unit often unfeasible. To cope with this, PDES has been put in place, in order to exploit the computing power offered by multiple parallel/distributed processors to carry out the simulation of the model.

While on the one hand parallelization of execution of events from different LPs was certainly an improvement regarding simulation performance, on the other hand it raised a synchronization problem regarding the correctness of the execution of events. In fact, given the independent processing of events belonging to different LPs, causality errors might be generated.

As an instance, consider the situation depicted in figure 1.2.

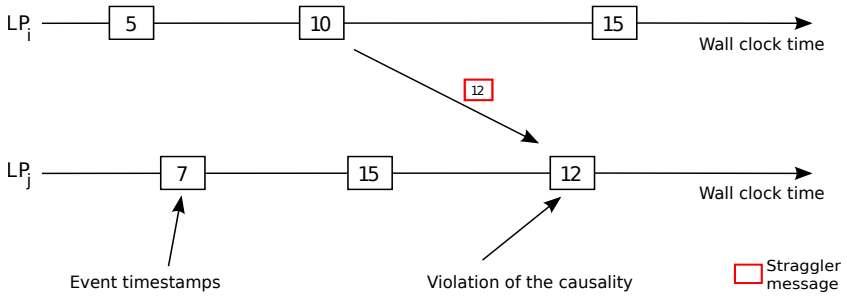

Figure 1.2: Causality violation example

Here, we can see that LP_i is sending a message associated with timestamp 12 to the LP_j, while the latter logical process is ahead in time (it already processed an event with timestamp higher than 12). In this case, the system needs a procedure to face this error, since the event that generated the violation, which is called *straggler* message, needs to be processed at LP_j before the event timestamped with the value 15. In general, a violation happens at an LP_k if it receives an event E_x with timestamp $T_x < LVT_k$.

To cope with this kind of situation, three main different synchronization approaches can be adopted:

Conservative synchronization mainly based on the avoidance of causality errors by building a structure aimed at executing only those events considered *safe*

Optimistic synchronization where execution is carried on without considering the possibility of causality violations generated by the processing of an event (namely, not considering its safety). In this case, a technique to revert to a consistent situation needs to be implemented.

Hybrid synchronization aimed at understanding which, between the previous two strategies, is the best appropriate approach to undertake regarding the running simulation model and choose it.

These three possibilities, largely discussed in the literature (cf. [Fuj89b], [Fuj93] and [Rey88]), are presented in the following subsections.

1.2.1 Conservative Synchronization

As stated above, conservative synchronization's goal is to avoid the occurrence of an erroneous situation. This was the first kind of synchronization that was introduced in the context of PDES, and at the same time the easiest one. In particular, the main idea at the basis of this approach is the usage of FIFO reliable links, that are by definition static. Every LP owns a predefined number of those channels, marked by a timestamp value (which can be either the timestamp of the element on top of the queue, or the timestamp of the last processed event). Then, every LP selects the next event from the queue with the smallest timestamp T_s among all possible queues. Also, note that if a queue Q_1 is empty and has a timestamp $T_{q_1} < T_{q_2}$, even if Q_2 is not empty execution should be blocked in order not to generate inconsistency. Indeed, extracting an element from Q_2 would be inaccurate since a new event E_k with timestamp $T_{q_1} \leq T_k < T_{q_2}$ could arrive, clearly generating a causality inconsistency.

Therefore, the system is able to select as the next event to execute the one with the smallest timestamp among all possible events, leading to execute a strictly *safe* event.

This technique, however, is prone to deadlocks. Consider, for instance, the scenario depicted in figure 1.3, where there are only 3 LPs in the system, each one of them collecting events in two lists (in the figure, an empty list is shown as blank rectangle). Given that all the logical processes own an empty list as the one from which they need to select events (being them the ones with smallest timestamps), execution is blocked for all of them until a new event is put in those lists. It is definitely a stall situation, for which many solutions were proposed. For instance, a possible workaround for this problem is represented by *null* messages, dummy messages not containing any event but used to notify other LPs that the sender, say LP_h, will not send any event e_x marked with a timestamp $T_x < T_{null}$. In this way, any blocked LP_j is free to execute events belonging to different queues, letting the execution to proceed.

Conservative synchronization presents thus some advantages and disadvantages, which are listed below:

– it needs a minimal synchronization among logical processes, because the wall clock time (simulation time) always increases (as opposed to *opti-*

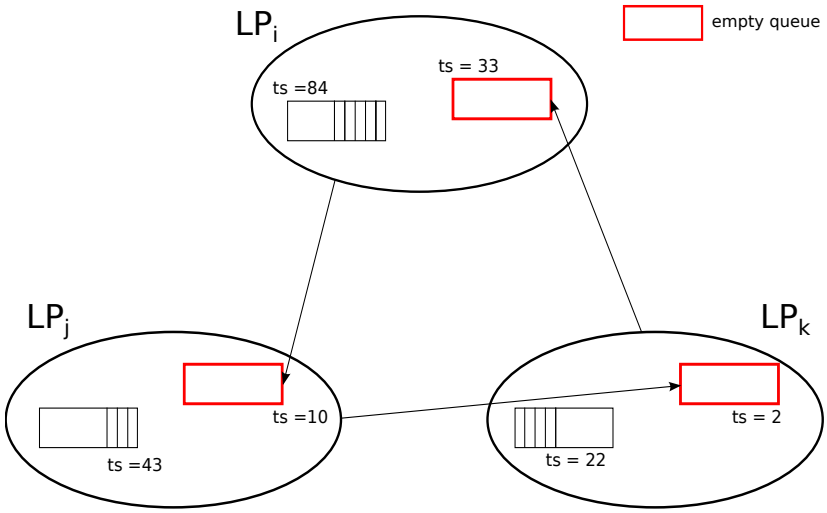

Figure 1.3: Conservative synchronization deadlock example

mistic synchronization discussed in the next section). This ensures that the GVT can be much easily evaluated.

– it is said to be aggressiveless, since the simulation goes on explicitly avoiding unwanted situations, namely not permitting, at each simulation step, to generate a causal violation.

– it is a kind of synchronization that is free of risks (riskless). In fact, no incorrect data is produced and thus it can be delivered to other portions of the model at any instant of time.

– it is not meant to effectively make use of parallel architectures. It is a common situation, regarding this kind of synchronization, the one in which the execution of two events could be forced to be serial, even if they are not directly related each other.

1.2.2 Optimistic Synchronization

In order to describe optimistic synchronization, the *Time Warp* technique, presented in [Jef85], needs to be introduced. This mechanism is mainly based on the idea that, as completely opposed to conservative synchronization, execution is carried on independently of the correctness of the executed events. However, it is able to *i*) detect if an inconsistent situation raised *ii*) keep the information introduced by the event that generated the violation *iii*) restore a correct state from which to restart the execution. This mechanism is strongly related to the

speculative processing, which consists in executing tasks that are evaluated as likely to be right. The great advantage of this kind of processing is the fact that, if the assumption was right, then the cost that would have been necessary in order to compute the correctness of it can be reduced to zero while, on the other side, if the inference was incorrect, the cost to come back to a consistent situation might be the same as if no speculative processing was carried on.

In the specific case of simulation, optimistic synchronization needs a method to come back to a consistent situation, if a *straggler* message was received by any LP. To have a better idea of what is the situation to cope with, let's refer to figure 1.4.

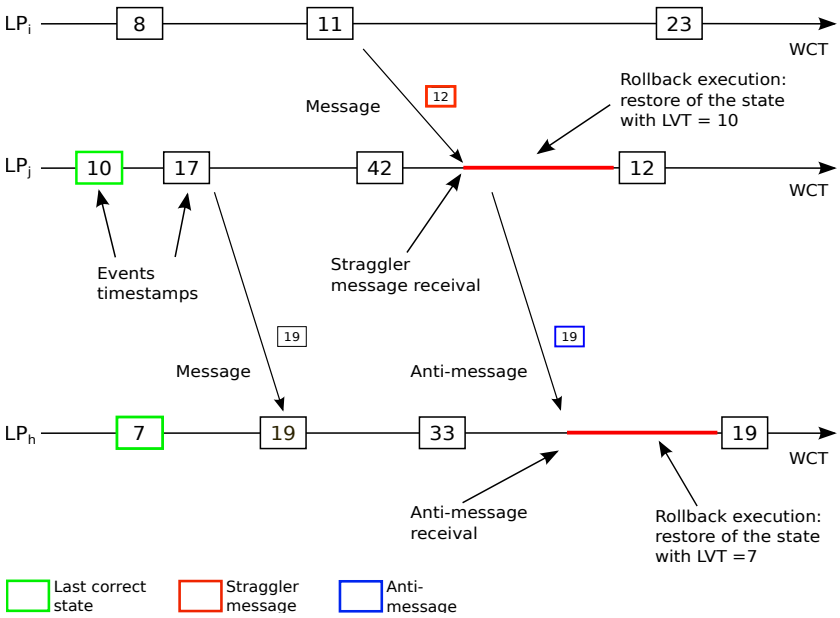

Figure 1.4: Example of execution where rollback is needed.

In the picture, the reader can see that LP_i, being it free to go ahead in execution without caring about causality, is sending to LP_j a message timestamped with the value 12, which is less than the last executed event's timestamp value at LP_j. Thus, the latter LP needs to come back to the state of the event having the closest timestamp to the one of the straggler message, in this case 10, since up to this pont execution was correctly carried on. However, LP_j at time 17 already sent a message with timestamp 19 to LP_h, and rollbacking to 10 would mean to undo also that operation. This is why, during a rollback operation, the usage of *antimessages* is needed. The function of this kind of messages is to annihilate the corresponding positive event, both in case it had been already executed or not. In the former case, a previous consistent state has to be re-

covered thanks to a rollback operation; in the latter, the positive event can be simply deleted from the event list. Optimistic synchronization presents many disadvantages:

- *high memory usage*
- *additional overhead*
- *aggressiveness*

and advantages:

- *high exploitation of parallelism*
- *better performance degree*
- *avoided usage of dummy values*

First of all, in order to perform a rollback phase the system needs to remember the states from which it is safe to restore the execution. This has of course an impact on the overall memory usage, which can represent a problem when the number of LP increases. Indeed, considering the work in [McK04], we can see that the *memory wall* limits the throughput even on most recent architectures (thus leading software manufacturers to move towards a distributed approach). Moreover, checkpointing, rollbacking and synchronizing bring additional overhead in the normal procedure of simulation, that in some pathological situations causes performance degradation. Finally, as opposed to the previous scheme, optimistic synchronization is aggressive, meaning that there can be incorrect state configuration forcing the system to rollback. On the other hand, it can be noticed that the execution of events in this kind of synchronization is never enforced to be sequential. In fact, no check about consistency is made and events are executed as they are extracted from their specific list. This definitely has an impact on performance when running with a parallel computing paradigm, as operation can be concurrently completed on various cores/machines. Then, regarding performance, the simulation trajectory is never interrupted due to uncertainty about correctness: this speeds up the execution when no violations are detected. Also, the model designer doesn't need to care about checking the consistency of the execution. Being the rollback operation only related to the system, it will be carried on when a dangerous situation is (automatically) recognized. This also leads to the avoidance of the use of *null* messages, since it is sufficient the analysis of the information of an LP to guarantee correctness.

Since rollbacking is a core operation when dealing with this kind of synchronization, many different implementations were proposed, and they will be discussed in section 1.3. Finally, note that it is quite important, regarding this approach, the agreement of LPs on a value before which every entity is sure that

execution has been finished and memory dedicated to checkponting it can be released (performing the so called *fossil collection* operation), namely the GVT (for further informations about how the possibly distributed GVT value can be calculated, please refere to [Bel90], [LL89] and [TPQ+17]). Consequently, between two consecutive values of GVT, an optimistic execution time window is defined. Nonetheless, the calculation of this value needs to be precisely pondered: if the time window stretches to a large value, the system tends to loose the benefits of this approach falling back to a high memory demand approach, while if it shrinks to a too small value, then LPs will need to run the GVT calculation operation too often, therefore increasing computational costs and coming back to the case of conservative synchronization. Since this evaluation can be strongly related to the underlying application model, different solutions are possible. In particular, observing the behavior of the system until a specific instant of time, it can be dynamically computed exploiting machine-learning algorithms and properly tuned as long as simulation is running ([Gos03]).

1.2.3 Hybrid Synchronization

The synchronization schemes described in the previous section are intrinsically characterized by some limitations and, as already stated, the application model developer should choose the best suited among them with respect to the phenomenon to simulate. However, there can be situations in which a mixture of the previous two alternatives could represent an interesting choice.

Hybrid synchronization (introduced in [RAT93]) works by tuning, as far as possible, both the conservative and the optimistic approaches, in order to insert in one some benefits of the other. In particular, two main possibilities are proposed:

- *Restricted conservativeness* which tends to avoid the system to be totally conservative, thus enabling speculative execution only locally, that is only for those events regarding a specific entity and not involving other LPs. This leads to the elimination of cascade rollback scenarios, and clearly reflects in a gain in performance. In this way, the *aggressiveness* mentioned above is reduced.

- *Limited optimism* built in order to get rid of the high memory usage and the possibly high risk degree typical of optimistic synchronization, defining a *commitment horizon* before which LPs cannot rollback. This means that the system allows LPs to execute events falling in a specific time window, and to agree on the GVT value in order to free up memory. Particularly, this approach is related to the GVT value considering its distance *in the future*: if an LP executed speculatively and reached a simulation time too

far from the last GVT calculated value , its advancement is slowed down (e.g.: *throttling*).

1.3 Rollback Strategies

Concerning synchronization alternatives, we already saw that when simulation is carried on in a speculative fashion if at a specific simulation time a causal violation is detected, a technique to restore a consistent state is needed. This means, practically, either to remember diverse undertaken steps during execution, or to elaborate a technique to execute backward the involved events.

1.3.1 State Save and Restore

The first solution, called *state save & restore*, basically consists in taking snapshots of simulation states of logical processes including their timestamp and a set of variables related to the specific LP. Depending on the mode used to perform this (costly) operation, we move across three different solutions: *Incremental State Saving* (ISS), *Copy State Saving* (CSS) and *Sparse State Saving* (SSS), which are discussed in the following subsections. A thorough description of this strategy can be found in the seminal papers [Jef85] and [Jef90].

1.3.1.1 Copy State Saving (CSS)

Basing on the idea of *state save & restore*, an immediate solution is the one of taking a copy of the simulation state of an LP as soon as an event is scheduled. This copy is marked with the timestamp of the event processed immediately before taking the snapshot. Thus, if a straggler message arrives at timestamp T_s, the system just needs to restore the state from the checkpoint with the highest associated timestamp value among the ones strictly lower than T_s. This is obviously a task as easy as costly, since repeatedly taking a *photo* of the system at a specific instant of time means asking, when time grows indefinitely and the number of LP is huge, great amounts of both memory and computational time. For this reason the aforementioned technique of *fossil collection* tends in this case to be necessary: considering the limitations of nowadays machines with respect to the demand in the context of simulation, it is crucial to free up some resources when reaching trashing thresholds. In figure 1.5 a rollback situation of this kind is presented.

1.3.1.2 Sparse State Saving (SSS)

The method previously exposed has been improved taking care of the high resources requirement needed. Basically, *Sparse State Saving* (SSS) aims at

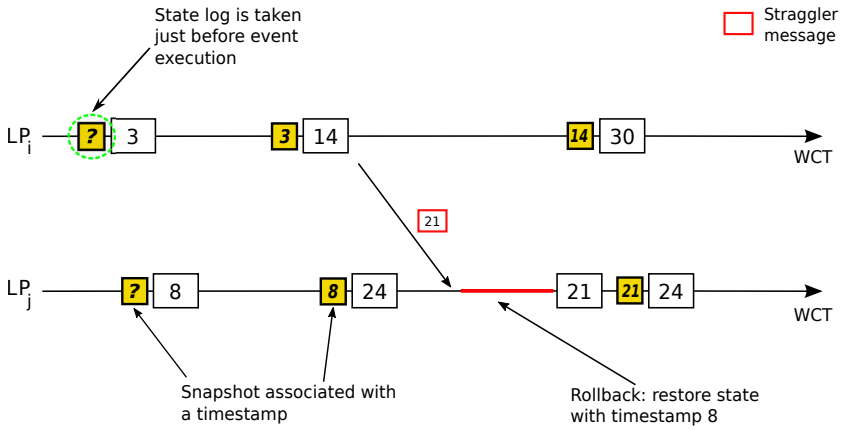

Figure 1.5: Rollback operation in the case of CSS

reducing the frequency at which the snapshots are taken, since it is not strictly needed to perform that operation at every event scheduling. The term *sparse* refers to the fact that state saving procedure is done in different instants of time, either periodically or heterogeneously.

When the capture of checkpoints is done in a periodic way (cf. [Bel92]), the name of this technique switches to *Periodic State Saving* (PSS). The main drawback of this kind of organization is the fact that at some point, when a rollback is necessary, there can be a lack of logs from which restart the computation. Thus, simulation needs to start from the latest secure photo of the LP's state with respect to the rollback's timestamp, giving rise to the so called *coasting forward execution*. In fact, if the last useful log was placed before some already executed events, those events must be processed again (thus rebuilding their states), keeping in mind that all the messages which may have been generated by those events directed to other LPs must be ignored, hence executing in a *silent* mode.

As an example, refer to the scenario depicted in figure 1.6. Here, state saving is performed every 3 events (periodically), but LP_j receives a straggler message (event's timestamp equal to 14, while LVT_j is 24) in the middle of his checkpoint period. Therefore, it needs to come back to the last available state, that is the one marked with value 8. Note that, while executing the event with timestamp value 11, LP_j sent a message to LP_i in order to make him execute an event marked with value 14. When re-executing in a silent mode after having rollbacked, LP_j avoids sending the same message again to LP_i, as it would be a backward event for it.

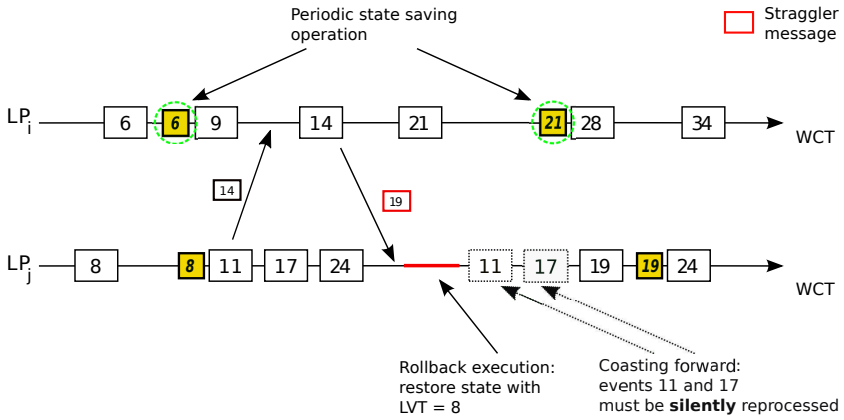

Figure 1.6: Rollback operation in the case of SSS

1.3.1.3 Incremental State Saving (ISS)

The final technique related to state saving is called *Incremental State Saving* (ISS), and it tackles the problem of high memory consumption related to checkpointing (as also faced in [DF97] and [DFP+94]), but from another point of view with respect to SSS. In fact, the main idea at the basis of this method is not to create, when planned, a completely new state snapshot but to memorize only those informations that were not stored in the previous log. Thus, only interesting data is, as the name suggests, incrementally added to the future snapshots. In particular, in order to perform so, events were augmented with additional informations. If, on the one hand this could be a drawback, on the other it drastically reduces the overhead generated by copying and restoring states in checkpoint and rollback phases, respectively.

A recent work presenting one of the most innovative approaches in the field of ISS can be found in [PVQ15].

1.3.2 Reverse Computation

The second solution, named *Reverse Computation* (RC), is instead based on undoing all the past events until a consistent state is reached, trying to eliminate, or at least to significantly reduce, the usage of checkpoints. This can be done exploiting compilers facilities, in particular generating, for each event e_i that needs to be undone, an *anti-event* which executes the same operations performed by e_i, but in reverse order. If the rollback request at LP_i with timestamp $T_{rollback}$ involves multiple events, the undo operation has to be done for each event with timestamp greater than the rollback instant, i.e. starting from LVT_i up to $T_{rollback}$.

As an instance, consider the following code snippet modeling a multiplexor ATM model cell transition (cf. [CPF99a]), resembling a possible event action:

```
if(value > 0){
    value--;
    count++;
}
```

It is easy to understand that the *anti-event* that would be generated for this particular case would have this reverse actions:

```
if(value "was" 0){
    count--;
    value++;
}
```

Although the inverse computation of arithmetic operations can be easily performed, the same procedure when encountering a branching condition may be not such an obvious task. Indeed, when reversing a branching condition it is essential to check an old value of a state variable (hence the "was" keyword in the above code block), which is not available during the execution of the reverse event. Thus, it results to be useful the introduction of a so called *bit variable*, a (transparently added) boolean variable which tells the system whether a branch was taken on not while normally executing.

```
if(value > 0){
    entered = 1; //bit variable: the branch was entered
    value--;
    count++;
}
```

When executing backward events involved in a rollback, the *anti-events* can make use of the value of this particular kind of variables to discriminate the cases in which a branch needs to be taken or not:

```
if(entered){
    count--;
    value++;
}
```

Anyhow, this solution introduces an extra state variable definitely increasing the size of LPs states. However, this size increase is quite negligible, since if the number of branches is n, then the amount of added variables would be $\log_2(n)$.

The same reasoning can be done for `while` statements, keeping in mind that the number k of iterations is a mandatory value to remember when re-executing in the opposite order the same loop. Additionally, note that not all the operation can be undone via reverse computation: a particular class of them, the *disruptive operations* (e.g. assignments or bit-wise computations) must be managed with state saving techniques, since they produce changes in the system that are note reversible.

Another important point to take into account is the handling of jump instructions, such as `goto`, `break` and `continue`. In fact, it could be a non-straightforward process to restore the actual execution flow. Again, the use of *bit variables* is fundamental, as the steps attempted during forward execution are to be remembered in some way. Moreover, for this particular case the aforementioned technique can be augmented by the (automatic) insertion of `switch/case` statements, in order to be able to execute the same flow in the reverse order. Nevertheless, this time the additional overhead caused by the added code portion can represent, depending on the complexity of the code to restore, a non-negligible state size increase.

Reverse computation can be considered an interesting alternative with respect to state saving, as it provides a faster execution of rollback operations with a minimal amount of overhead, especially in situations in which the rollback target point is near in time to the current logical time of the involved LP. However, the more the number of *disruptive operations* grows, the more it moves back to the fine-grain state saving solution.

Finally, since this kind of solution is not adopted in the work presented in this thesis and has been studied elsewhere, we refer the reader to [CPF99b].

1.4 PDES Logical Architecture

In the last years, the high demand of computational costs with respect to simulations, which spread among many research fields (such as demography, engineering, medicine, etc.) brought to the design and development of systems and architectures following the parallel and distributed computing paradigm. It is well known, in fact, that dividing the whole amount of load among different cores on a local machine, and among different machines in a cluster can strongly increase performance, provided that coordination and cooperation between nodes are properly implemented.

As far as DES is concerned, this kind of architectures have been proved to be well suited to get better performance, leading to the concept of PDES. In particular, being DES characterized by a high number of LPs interacting with each other thanks to timestamped messages exchange, the great advantage came

with the possibility of processing different LPs' events on different processors in a concurrent way and, moreover, to distribute the LPs on different machines communicating via an interconnection network (e.g.: LAN).

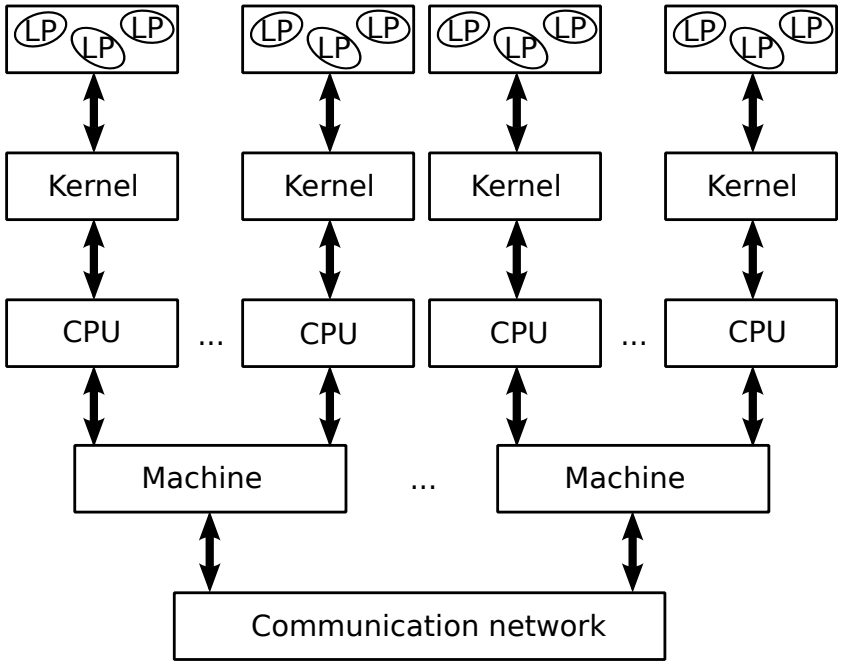

Figure 1.7: PDES classical architecture

PDES architectures are also based on the idea of *simulation kernels*, which are user-space processes running on top of a host operating system and thus related to a specific CPU. The model interacts with it thanks to minimal interfaces, notifying the creation or the execution of an event, or that an ending condition was reached. Simulation kernel instances manage a set of LPs, and can communicate with other instances either exploiting facilities provided by the O.S. they're running on (e.g.: *Inter-Process Communication*, IPC), or by taking advantage of message passing primitives (e.g.: *Message Passing Interface* protocol, MPI). An important peculiarity of this organization is the fact that,in general, LPs may or may not directly access other one's state, even if they lie on the same simulation kernel. In fact, on some platforms this kind of operation is only supported, again, via message exchange, and that's an important point to stress since it will be discussed in the remainder of this thesis. Figure 1.7 shows an example of this type of architecture.

With the advent of multi-core and *Symmetric MultiProcessing* (SMP) systems, PDES architectures evolved to a new structure where basically less in-

stances of simulation kernels are deployed on a single machine, handling a higher number of CPUs. Therefore, kernels rely on multiple worker threads managing any LP. This kind of architecture is presented in figure 1.8

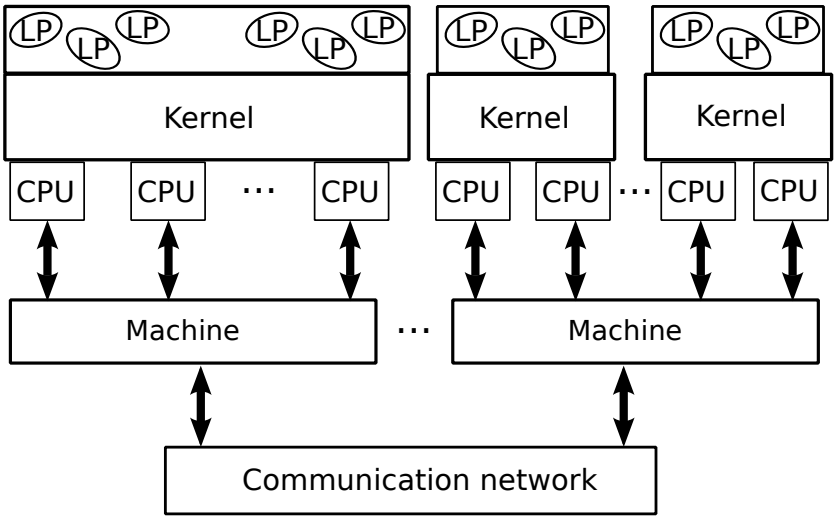

Figure 1.8: PDES newest (multithreaded) architecture

As a final note, given the possibility of a worker thread to manage whatever LP, the cost of switching a logical process from a thread to another should be considered, especially if the interested threads lie on a remotely connected simulation kernel instances. This is because migrating an LP_i means to transfer all the data related to it, including its simulation state S_i and its event queue(s). To this end, typically a LP binding is defined, in order to avoid transferring objects among worker threads too often, consequently degrading performance. This binding is intended as a time interval inside which the operation of moving a LP is not permitted, and can be reduced or enlarged discretionary. However, LP migration is not always an operation to bypass: it can be very useful when the system recognizes that the load among processing units/computing nodes is not evenly distributed enough. In fact, migration can be used to implement a load sharing technique, when dealing with a distributed environment. Finally, a trade-off between the two concepts should be found.

1.4.1 The ROOT-Sim Environment

The whole project developed and presented in this thesis was realized on top of an Open Source, multithreaded, distributed general purpose simulator called **RO**me **O**p**T**imistic **Sim**ulator (ROOT-Sim). It is a simulation platform writ-

ten following the C/POSIX standards, developed according to the Time Warp protocol and targeting the x86-64 architectures. Being it created in 1987 by the High Performance and Dependable Computing (HPDC) research group at Sapienza, University of Rome it now implements the most recent research discoveries regarding optimistic PDES, also supporting distributed computing making use of the MPI protocol.

From an architectural point of view, ROOT-Sim is constructed thanks to a series of building blocks interacting with each other, as depicted in figure 1.9

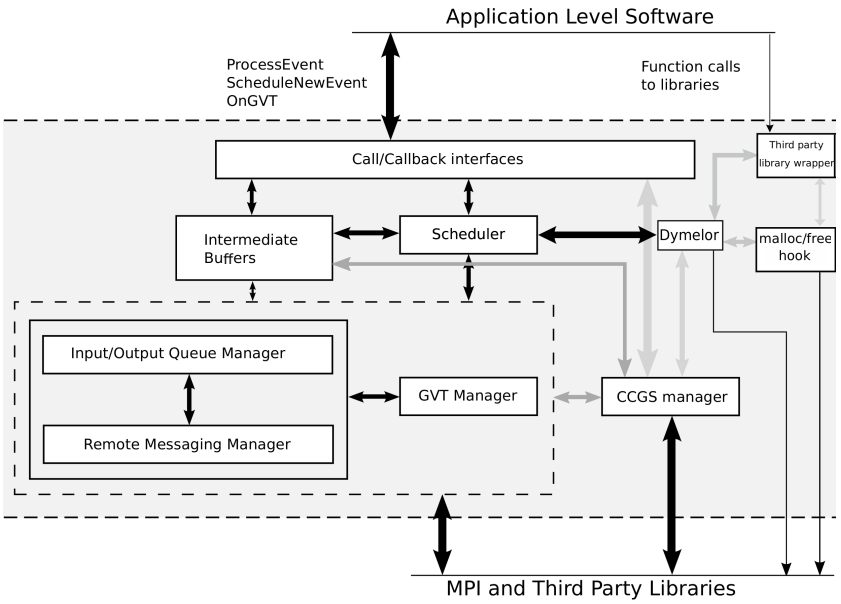

Figure 1.9: ROOT-Sim architecture

In particular, the application level software, (i.e.: the code representing the model the user is interested to simulate) is interacting with this environment through three main APIs: `ScheduleNewEvent()`, `ProcessEvent()` and `OnGVT()`. The first one can be used to notify the simulation kernel that a new generated event is ready to be scheduled. The kernel will process the request delivering it to the correct recipient, who is in charge of scheduling it according to the adopted scheduling function. The second is the function called to request the processing of an event. This is why it accepts as parameters some values representing the event's details, such as sender, receiver, type, content and so on. Finally, `OnGVT` offers the simulation kernel the possibility to notify the model that every LP evolved their simulation until a committed state, that means a state before which they cannot rollback. This is a core function since thanks to it the user can understand, evaluating a sort of global predicate, if

the simulation reached the ending condition.

The most important sub-systems this architecture is composed of are the following:

- **Dynamic Memory Allocator (DyMeLoR)** This library, proposed in [TQ08]), is basically a wrapper for the widely spread `malloc/free` function calls, and in particular it was developed to better manage memory regarding state saving. *DyMeLoR* works essentially by preallocating equally-sized chunks of contiguous memory, upon a memory request made by the model. To keep track of what chunks are currently in use, a per LP meta-data table called `malloc_area` is maintained, storing the reference to a bitmask that keeps informations about a block of chunks. After building up those data structures, *DyMeLoR* finally calls the `malloc` function. This wrapper evolved in time to *Di-DyMeLoR* (cf. [PVQ09]), which scope was to offer a better memory usage by storing, step by step, only state differences, exploiting the incremental approach presented earlier.

- **Scheduler** This subsystem is aimed at selecting the next executing LP. Thus, it defines an order among all the LPs in a simulation kernel. Within ROOT-Sim different kinds of scheduler were implemented, but as a default the *Smallest Timestamp First* (STF) one is preferred. As the name says, this scheduler activates the LP whose next event to execute is the one with the smallest timestamp among all queues of all LPs.

- **Global Virtual Time Manager** As described in the previous sections, this module is delegated to periodically calculate a GVT value in order to let the LPs free up memory.

- **Committed and Consistent State Manager (CCGS)** This block is designed to periodically try to establish if a terminating condition was reached. It is strongly related to the GVT manager, since it uses the GVT calculus also to start the termination procedure. For instance, in some cases it is enough to check if a certain amount of time elapsed.

ROOT-Sim implements the aforementioned *worker thread* paradigm. In particular, this platform supports both multi-core machines, thus enabling (multi-threaded) parallel execution, and distributed computing. Inside this architecture, every logical process is managed by a worker thread and it's identified by *i*) a unique *Local Identifier* (LID) on the machine it is running on *ii*) a unique *Global Identifier* (GID) among all the distributed machines interacting each other. Every worker thread manages a pool of different LPs and can be bounded to a specific physical core parallelizing the workload over it. Threads schedule LP implementing the DES skeleton presented in section 1.1.1. Each

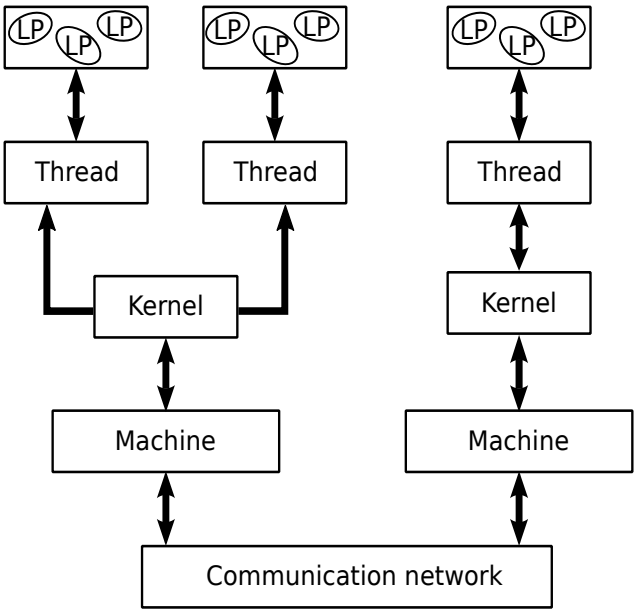

Figure 1.10: ROOT-Sim example of possible configuration

of them is identified by a *Thread Identifier* (TID) on the node it is executing. Even if it is not a strict rule, typically a single simulation kernel instance is deployed on a specific machine. Kernels regulate multiple threads, maintaining their data structures and handling inter-kernel/inter-machine communications. For further informations about this platform, please refere to [PQ14a].

Chapter 2

PDES Programming Models

Starting from its birth, the great advancements in high performance computing, the beginning of the multi-core era and the spread of rich distributed computation resources lead Discrete Event Simulation to constantly evolve and tune to fully exploit the platforms it needs to rely on. In fact, research in this area widely increased over years and many researchers started to care about how to deal with this new types of platforms. In this chapter, we present an overview of the steps research moved to bring simulation at the point it currently is, starting from the very beginning (sequential DES) to the newest results (distributed PDES). In particular, we want to focus on how memory was intended to be managed from an implementation to another, in order to enlighten the benefits and the disadvantages of different approaches, delivering the reader an exhaustive analysis of the state of the art that also made us taking the choices discussed in this thesis.

2.1 Sequential and Distributed DES

The very first implementation of DES was inherently sequential. The main idea behind this (as we outlined in the previous chapter) was to have an *event queue* keeping the future events to be executed and a *clock*, holding the simulation time (WCT). At each time step, an heuristic was used to select the next event to execute, that typically was the *smallest timestamp first*. Thus, after simulating an event, it was removed from the queue and it could either generate a message to be inserted in the queue or cancel a previously scheduled event. Given that the whole simulation is based on the execution of events causing the advancement of the *clock* variable, we call this approach *event driven*. Note

that there exists another approach, called *time driven*, in which simulation is carried on by increasing the *clock* variable by one tick at each simulation step, executing the events scheduled for that time.

Regarding this sequential paradigm, the reader can see that this is not easily parallelizable: only one event is extracted from the queue at each simulation time and its consequences are possibly reflected, thus not taking advantage of other cores on a machine, provided that the *event list* cannot be properly partitioned in such a kind of execution. This is clearly a key problem since nowadays architectures are, from the lowest to the highest end, equipped with a number of CPU/cores. Thus, the first trial of exploiting this kind of organization was represented by the definition of (sequential) distributed DES. Briefly, it was at first based on spreading the just discussed idea on multiple CPU/cores (recall that being distributed doesn't strictly mean to execute on different physical machines). Of course, even if concurrency problems are still avoided, the introduction of communication between simulation entities was needed, giving rise to problems regarding the sharing of the memory we want to discuss in this chapter. In fact, it is important to notice that this kind of communication can be accomplished according to three main different approaches:

- **Share everything** any data item declared inside the memory block dedicated to a LP can be accessed by any other LP running in the system. Note that the distributed nature of a simulation platform is perfectly inline with this kind of organization (think about data replication/mirroring).
- **Share nothing** any LP is not allowed to access any memory portion among other LPs. In this organization, communication is only supported via message passing. Implementations of this organization purely rely on messages exchange or Remote Procedure Calls (RPC). Consider, however, that it can be implemented also in a shared memory machine.
- **Hybrid sharing** the system determines and restricts the amount of shared data that entities can access in a shared way.

As far as the above problems are concerned, when dealing with processors, one important point to discuss is their coupling. Indeed, if two processors are loosely coupled, they can execute in an independent way, without conveying with others, for a longer time period, thus well fitting the *share nothing* approach. However, typically this is achieved via a pre-processing phase, in which static data partitioning, a not always trivial operation, is performed in order to properly employ each entity and also to compute some relevant simulation parameters. On the other hand, if the processors are tightly coupled, then the high amount of event execution by LPs would mean a great amount of messages to be exchanged between processors, originating a significant additional overhead and leading to

the choice of the *share everything* mode.

2.2 PDES Strikes Back

With the advent of multi-processor/multi-core distributed high-scalable architectures, the sequential execution of DES became computationally intractable. However, even if the implementation moved to a parallel version (PDES), the main (memory) problems discussed above still remained interesting issues to cope with, being them easy to directly map on those kind of architectures. In particular, PDES simulations rely on *parallel simulation languages* (PSL), capable of exploiting the possibilities offered by the underlying software (i.e.: thread definition and management, IPC, etc...), usually based on a *share nothing* memory programming paradigm. This represents an active research area, even if literature presents many works related to this problem which is interesting to discuss.

Traditionally, the problem of memory accessing by various LPs during simulation was faced according to the approaches described above and the most important proposed solutions rely on:

- *Shared variables*: this covers the case of *share everything*, in which (a part of) the state of an LP can be accessed by multiple LPs, namely accessing shared (global) variables resembling the wide spread sequential-style programming.
- *Message-passing*: as opposed to *share everything* approaches, this is mostly built on requesting the access to specific memory areas, instead of directly accessing it as in conventional shared memory systems. This clearly involves a messages exchange-oriented protocol.
- *Mixed sharing*: when implementing memory accessing combining notions of both *share everything* and *share nothing* approaches, we come to a solution of this kind. Either partitioning shared memory, or adapting the system to a well suited approach can represent, in some cases, the best performing choice.

Note, however, that the discussion carried out so far and the results listed above may be implemented in both *conservative* and *optimistic* simulation approaches ([Fuj89a]). In fact, they can be properly tweaked in order to support either rollback strategies or *safe/unsafe* event discovery. Given that the speculative paradigm is the one preferred in this thesis, we will only focus, if not differently stated, to results related to it. In the following subsections, we briefly present the aforementioned solutions, along with some details of implementations that literature provides.

2.2.1 Shared and Global Variables

Before going through the implementation problems and development of them, let us discuss what is the goal of shared variables in the context of PDES. In order to do so, consider the simulation example provided in [GF91] about battlefields. Here, a rectangular war field is divided in a number of hexagonal sectors equal to the number of LPs, each of them in charge of managing its area. In order to carry on simulation, each time a combat unit moves from a sector to another LPs need to know some information about them (e.g.: strength, amount of soldiers...) in such a way to make decisions about the next action to take (namely retreat or make an attack). If shared memory facilities are provided in such a kind of scenario, then modeling and executing it could be a straightforward operation. The realization of this approach in PDES is however related to some problems. First of all, a well known property to cope with in a parallel architecture is consistency. Indeed, if the interested value(s) is implemented as a global variable, then the system needs a protocol to maintain a consistent version among copies of it on different CPUs/nodes. Moreover, DES introduces the concept of *virtual time consistency*. Consider, as an instance, the situation in which LP_i wants to update a shared variable V_x at time T_i and LP_j needs to read the same value at simulation time T_j, being $T_i < T_j$. Suppose, also, that $LVT_i < LVT_j$, and that LP_j reaches the read operation of V_x before the write operation issued by LP_i. In this way, LP_j won't see an updated value of V_x event if the LVT at which it reads is logically greater than LVT_i. This is clearly a situation in which virtual time consistency is not kept. A possible and popular solution to this issue is the definition of an *owner* LP, which stores a multi-version list for each shared variable storing the history of the values assigned to it at some specific LVT. This is done because, as rollbacks can occur, if a LP reads a value which is later withdrawn, it needs to roll back as well to a consistent version of the variable. On the other hand, when a LP generates a new version of the shared variable, a new entry in the version list storing the new value along with the LVT of the writer is inserted. As an important note, consider that a parallel execution of such read/write instructions can generate concurrency problems, which is why lock-based programming is mandatory. Indeed, this can represent a bottleneck when implementing such a kind of architecture, and may lead, in some worst-case scenarios, to not finally having a considerable gain in performance. The work in [PPQV16] presents a solution similar to the described one, which fully exploits multi-threading paradigm benefits via per-thread memory global variables. Accesses are transparently intercepted at run-time and concurrency problems are tackled making use of ad-hoc non-blocking (wait-free) data structures.

Another similar approach to the one just presented is proposed in [GF91].

Here, the organization of shared variables is focused on the idea that memory portions are associated with an address, as the standard view of the memory, and to a certain timestamp, expressing the version in time of the current block. Coming back to the battlefield example, situations in which different LPs demand a snapshot of the battlefield grid at different simulation times often arise. Hence, in the proposed scheme, a state shared variable can ideally be plotted in a chart where the coordinates are its memory address and its LP simulation time (hence the name *Space Time Memory*, STM). Whenever a LP desires to access such a variable, the entry corresponding to its timestamp is returned. Algorithms 2 and 3 show how writes and reads are managed in this proposal. In particular, every event stores in a list the versions, specific of an object, that it read over time, and the same is done for the written versions. Those lists are fundamental in order to implement rollbacks. Also, an earliest timestamp event, *EarliestW*, who is currently writing its version of a variable is kept. On a single object a pair of locks are defined: earliest writer's time stamp (EWTS) and write lock (WR).

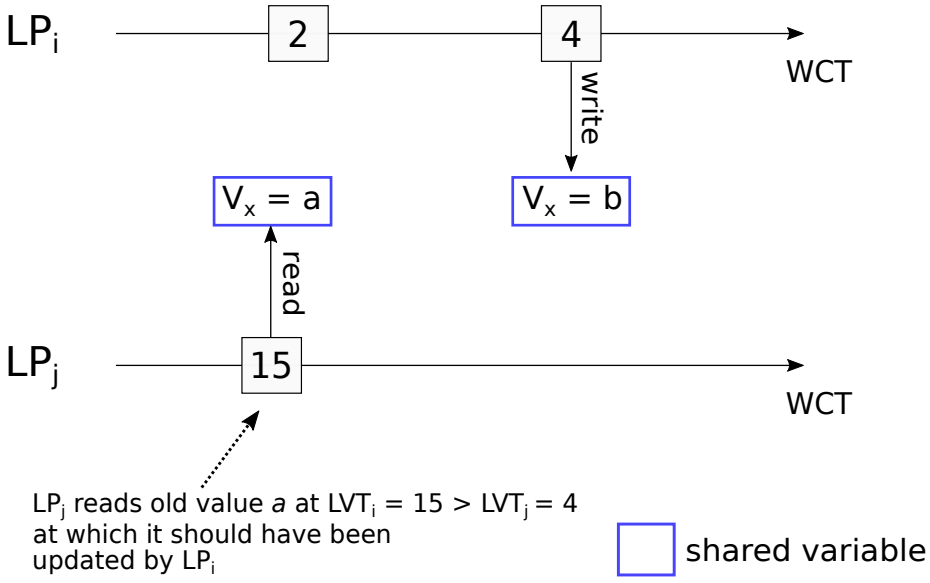

Figure 2.1: Virtual time consistency violation.

Algorithm 2 Space Time Memory Read Operation

```
1: procedure STMRead(Obj, EventTimestamp)
2:     while EWTS lock not obtained do
3:         nothing
4:     end while
5:     if EventTimestamp < EarliestW then
6:         while WR lock not obtained do
7:             nothing
8:         end while
9:         release EWTS lock
10:        entr ← Obj list entry with higher timestamp ≤ EventTimestamp
11:        if entr == NULL then
12:            raise error
13:        else
14:            release WR lock
15:            return pointer to entry
16:        end if
17:    else
18:        release EWTS
19:        insert event in obj's read/write waiting queue
20:    end if
21: end procedure
```

Algorithm 3 Space Time Memory Write Operation

```
1: procedure STMWrite(Obj, EventTimestamp)
2:     if no free entry to write available then
3:         free up space via fossil collection
4:     else
5:         while EWTS lock not obtained do
6:             nothing
7:         end while
8:         if EventTimestamp < EarliestW then
9:             EarliestW ← EventTimestamp
10:            while WR lock not obtained do
11:                nothing
12:            end while
13:            release EWTS lock
14:            entr ← Obj list entry with higher timestamp ≤ EventTimestamp
15:            copy entr in new entry
16:            for each reader event e with timestamp < EventTimestamp do
17:                rollback e
18:            end for
19:            insert new entry of Obj with EventTimestamp in list
20:            invalidate entries e_i with timestamp > EventTimestamp
21:            rollback events accessed e_i
22:        else
23:            release EWTS
24:            insert event in obj's read/write waiting queue
25:            release WR
26:            return pointer to new Obj entry
27:        end if
28:    end if
29: end procedure
```

2.2.2 Message Passing

This kind of technique is quite orthogonal to the one of the previous subsection. Indeed, when the *share nothing* concept is to be put in place, this is the most spread realization that literature provides. In particular, the core idea is that the communication between logical processes which have the necessity to access each others' state is ultimately set up by a (possibly distributed) protocol. Depending on the approach the protocol is based on, we can distinguish two main kinds:

- *Pull-processing*
- *Push-processing*

To understand the rationale behind both of them, consider a set of n LPs, say $LP_1, \ldots, LP_n$, and a single LP, namely LP_x, which holds a specific portion of data V_x that has to be read by any LP before being updated. As for the first mechanism, when reading/writing V_x it works, respectively, by i) asking LP_x to send a copy V_x ii) requesting the current value of V_x to LP_x before having the possibility to update it, then change it and send back the updated value to LP_x. *Push-processing*, instead, works by replicating an update of certain fraction of data to every $LP_i, i \in [1, n]$ in such a way that each of them sees, at any simulation time, a consistent version of it.

At this point, it is easy to figure out that both the two methods require a high amount of messages to be exchanged in order to synchronize and preserve consistency. Indeed, in the first scenario, when the update operation occurs, at least $2n + 1$ messages are needed, while in the second approach, each time an update is performed in an LP in the system, a message needs to be sent to any other entity (in our particular case, n messages per update).

One of the most famous message-passing approach literature provides is the one presented in [FD97], where a Time-Warp based synchronization for state-dependencies is proposed. This work introduced the notion of Cross-State synchronization, which is the main topic of this thesis. Mainly, it works by using a message exchange pattern in order to support the update of LP states which are dependent from data belonging to different LPs. Being it designed to run on optimistic simulators, this protocol is able to cope with rollback situations, by means of *logically blocked* execution states when the requesting LP is asking data. The main advantages this work provided are related to an easier coverage of causality violation scenarios that may arise when shared data structures are widely used, where non-recoverable situations can happen.

2.2.3 Mixed Sharing

The arrival of new technologies, both in hardware and in software, produced an innovative rethinking of the problems DES needed to tackle. The advent of distributed programming paradigms (MPI and openMP), thread based parallel libraries, and Transactional Memory (TM) supports contributed to a new way of sharing the load among the all possible computation resources. As mentioned above, multi-threaded development techniques played an important role not only regarding parallel executions, but also concerning shared/private memory issues ([PPQV16]). Given the great heterogeneity of the platforms simulation researchers adopted, an hybrid exploitation of the aforementioned proposals was encouraged. In fact, it is common practice to adapt the development of systems to the underlying platform evolution, finding the best trade-off among the available possibilities. In this context, we mean with *mixed sharing* a different mode of operation with respect to resources (namely, memory) that fits aspects of both *share nothing* and *share everything* approaches.

The research trend, in particular in the area of memory-accessing, stepped forward to a different organization of logical processes aimed at gaining higher throughput by grouping them and exploiting the profits of collaborative execution ([MNPQ16]). Indeed, depending on the simulation model case, LPs can be partitioned in coarser or finer grains, as explained in [CLY+11]. In particular, this work introduces the notion of *Extended Logical Process* (Ex-LP), which essentially is a collection of LPs. The LPs forming an Ex-LP are allowed to access each others' states directly (thus resembling shared memory methods). However, those accesses cannot be traced by future possible rollbacks (which is a well known disadvantage also previously discussed), and that's the reason why every Ex-LP needs to maintain a processed event list holding all the events executed by LPs within the Ex-LP. In order to support such a kind of operations, different types of (state) variables were defined: *private*, which can be only accessed by members of the Ex-LP it is owned by; *public*, accessible by all entities in simulation, even if not belonging to the same Ex-PL; *common*, specifically related to the case in which the selected grain for grouping was the finest, i.e. simulation events are the parallel units, no LPs are used: in this case *common* (state) variables store all the simulation's state structures, and are normally accessible. To have a clearer idea of how the accesses are performed, see figure 2.2, where from left to right the possible grouping approaches are shown, from the coarsest to the finest grain. Here, different colors discriminate different Ex-LPs.

As stated above, to increase efficiency group operations are performed: in this case, rollbacks are undertaken on a per Ex-LP way, in order to avoid recurrent rollback cases that may occur among LPs in the same partition. It is important then to categorize LPs on the basis of some parameters related to the

kind of grouping is performed: in this particular case, close causal relationships was taken into account.

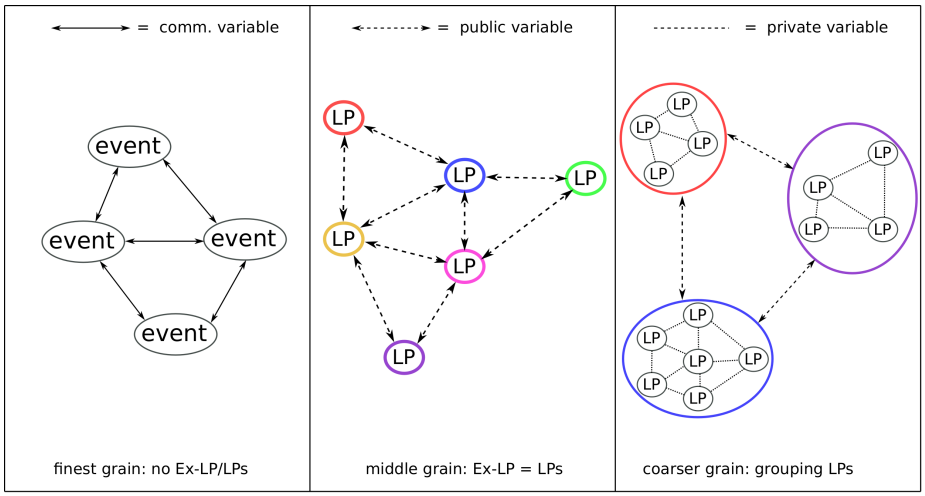

Figure 2.2: Ex-LP grouping examples.

As for other examples of hybrid technique, it is to be noticed that the *Event Cross State* solution, initially proposed in [PQ14b] and then augmented in this thesis can be seen as a mixed approach. Indeed, the ECS synchronization proposed in that work is based on *i*) a message-passing synchronization protocol and *ii*) kernel level based approach to access other entities' memory. The first point guarantees that the involved LPs all reach the same LVT in order not to generate consistency problems when reading/writing each others memory, while the second ensures that the involved LPs can directly access the memory of the LP they're targeting to. The innovative approach of this thesis doesn't exactly fit the categories mentioned so far. In fact, it is based on the concept of *leasing* of memory portions: an ephemeral (transient) owner of that memory portion is defined , which is in charge of managing it for a certain period of time. Specifically, the great advantage of this thesis, differently from all the other approaches presented in the literature and discussed above, is the fact that the programmer has the freedom of implementing a custom model following the pure sequential style programming paradigm. Indeed, the whole synchronization concerning the simulation state of an LP, which as mentioned above becomes completely shareable in an on-demand fashion, is entirely delegated to the run-time underlying layer. Thus, we didn't focus on global variables, which were widely addressed in the works presented above, since they represent an orthogonal problem with respect to ours. Details of our idea are presented in the next chapter.

Chapter 3

Distributed Event Cross State Synchronization

As described in the previous chapter, the constant need of a logical process to access and possibly modify other LPs' states brought research to introduce the concept of *Event Cross State* synchronization. With the software architecture presented in this thesis a transparent materialization of ECS on top of distributed memory systems is achieved. Thus we enable the exploitation of any kind of distributed resource for running the simulation model, while not sacrificing the flexibility of shared-memory accesses offered to the programmers by the ECS paradigm. Also, this proposal is based on fully transparent roll-back based optimistic processing, which has been proved to be a core building block regarding the scalability of the model execution. This solution targets the Linux operating system, in particular aiming at properly enhancing its memory management subsystem in order to:

1. detect, by pointer dereferencing in shared-memory application coding, the materialization of cross-state accesses among LPs hosted either on the same machine or on remote ones.

2. provide a mechanism to permit the accessing LP to transparently gain a lease on the requested logical pages related to the targeting LP's state, at the specific time of the event that triggered the synchronization.

3. discover on the fly what machine instruction actually generated the access, in order to get crucial informations (e.g.: which kind of operation it was, how much data involved) and to pre-fetch and locally materialize the proper number of pages that will be written back to the remote LP's state at the end of the cross-state interaction.

3.1 Reference System Model

In this section, we formally describe the reference system archiecture, which follows the organization already sketched in chapter 1.

A set of (possibly non-homogeneous) CPUs, scattered across any number of different computing nodes is supporting the PDES application model runs. Every computing node executes any number of simulation kernel instances, developed according to the multi-threading paradigm, where a shared-memory approach is used to enable intra-kernel synchronization. Inter-kernel communication is then supported by the underlying network interconnection linking all distributed machines.

In this organization, each simulation kernel instance spawns at startup as many worker threads as the number of processing units the system provides, and binds every thread to a specific CPU for a specific time window selected before starting execution. This binding, evaluated according to some *binding rule* and valid for a specific predetermined time window, represents an important feature since it ensures that for a certain period of time only one worker thread is allowed to schedule events related to a specific LP. Bindings can be recomputed either after the specific wall-clock time interval has elapsed or basing on some runtime parameters for the sake of an even distribution of simulation workload on the available computing power.

To have a more formal description of this simulation reference system model, we can state that it is composed by:

- **Kernel set**: a set (named $KernelSet$) composed by K different simulation kernel instances scattered across the available machines.
- **Thread sets**: K sets of concurrent worker threads, namely $TSet_k$ being $k \in KernelSet$, each of them exploiting the shared memory paradigm to accomplish their internal synchronization and communication tasks.
- **Logical Processes sets**: n sets of logical processes bounded to a specific thread. At any WCT in fact, a thread t, where $t \in TSet_k$, $k \in KernelSet$, is demanded to schedule events of a group of bounded LPs, namely $LPSet_t$. Note that, thanks to the binding rule, at a certain instant of time one LP is managed by one and only one worker thread, i.e $LPSet_i \cap LPSet_j = \emptyset \quad \forall i, j \quad i \neq j$

Also, considering the distributed nature of this kind of architecture, every LP sees, at any time, a set of *local* LPs, which are the ones running on its simulation kernel instance, and a set of *remote* LPs, in any other case. Formally, an LP_i considers as local all the LPs bound to any worker thread $x \in TSet_k$ s.t. $LP_i \in LPSet_t$ and $x, t \in TSet_k$.

As stated above, the first operation that this proposal is able to perform is the detection of the initiation of a cross-state synchronization. In particular, this is done by exploiting innovative kernel-level facilities, which are later discussed. Mainly, a page fault upon accessing the memory area dedicated to the state of a different LP is recognized to be the origination of a (possibly remote) ECS synchronization. This kind of synchronization relies then on a (potentially distributed) communication protocol, built on the notion of *control message*. This is a special kind of message, exchanged between two different LPs, since it is not included in the receiver's event queue as they are related to ephemeral state transition that are not to be repeated upon rollback procedures. Correctness of the protocol is enforced by two main aspects:

– Preemptable execution of the events of a LP

– State transition based on a well-formed state machine

To guarantee the first of the two points, simulation is based on the concept of *User Level Threads* (ULT). Those are basically CPU contexts that a thread $t \in TSet_k$ is able to suspend and restore at any instant of time. Specifically, in order to give control to a logical process, the worker thread managing it allows the execution of the events in an isolated environment, changing its CPU context and thus switching to a different stack. Hence, if the simulation platform takes control back and determines that an event's execution needs to be preempted, the worker thread just restores the CPU context related to its execution in platform mode and deschedules the running LP. Simply restoring the LP's CPU context, the execution of the suspended event can be resumed whenever the platform disposes it. Thus, it can be easily noted that having different memory areas dedicated to different execution entities can be a crucial aspect in executing events correctly. A technical description of the realization of this technique is presented in [PQ17].

As for the second point, the reader can see that the state machine counts three different types of states:

- *Blocked states*: those are the states referring to the LPs that are descheduled while executing an event, making use of the ULT facilities. A logical process associated with this state is not able to interact with other LPs, until simulation platform notifies it that a condition was reached, namely the reception of a specific kind of control message. This kind of states are grey-shaded in picture 3.1.

- *Ready states*: states related to those LPs that are able to either start the execution of a new event or to resume the execution of a preempted one, but that are not scheduled yet. In figure 3.1 they're white colored.

- *Running states*: as the name suggests, those are the states of LPs that are currently executing an event. Running states are green in picture 3.1.

The transitions from a state to another are either caused by *cross-state detection* (e.g.: page fault disclosure) or by actual *LP synchronization* (e.g.: control message arrival).

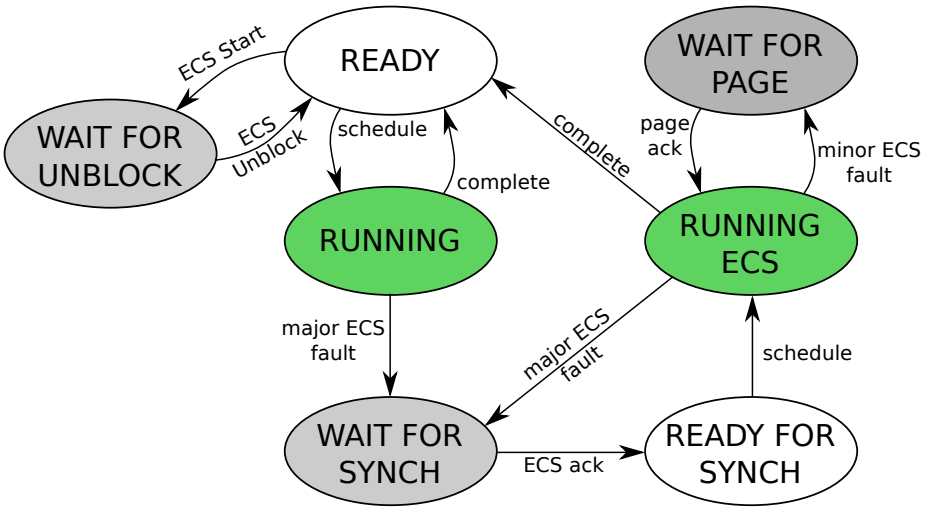

Figure 3.1: LP state machine.

3.2 Memory Management Policy

In order to provide a cross-state access detection, the system must enforce a memory management policy which easily enables to map a LP to a specific given memory address range, and viceversa. This is a core aspect because every LP has the necessity to understand if it's targeting a local or a remote LP's memory. Thus, the goal that we wanted to reach is to understand what LP's state is being requested by relying on pure address space mapping.

Moreover, when simulation is started up, there are multiple symmetric simulation kernel instances running on potentially distributed machines: this condition requires a sort of agreement across the different kernel instances upon LP state mappings over the same virtual address ranges. Also, as an another important point to stress, transparency was considered in developing such a kind of memory management. In fact, the application-level developer is not asked to

care about this agreement, as well as about dynamic memory allocations. Additionally, the arrangement across simulation kernel instances could be over-costly at runtime, which is why memory aspects are dealt with at startup time.

In particular, following the scheme presented in [PQ14b], the LP dedicated memory consists of a contiguous virtual block of 1 GB, referred to as *memory stock*. Every memory stock base address, as depicted in figure 3.2, is deterministically computed by every simulation kernel instance and is unique among all LPs living in the current execution. Thus, every simulation kernel instance maps LPs identically, meaning that the address range is defined also for the LPs that are not residing on a specific instance. Therefore, what was implemented can be defined as a *deterministic memory map manager*, resorting what was proposed in [PDQ11]. For instance, in figure 3.2, simulation kernel K_1 is mapping, inside its address space, stocks for LP_1 and LP_2, which it's going to host and manage (green colored stocks are the ones referring to *local*, with respect to kernel instance, LPs), but also for LP_3 and LP_4, which instead are running on another remote kernel instance, K_2. In this way, accessing a memory area falling outside the stocks of local LPs, i.e. going from green to grey shaded areas in picture 3.2, would mean trying to access a remote LP's state that is precisely what we wanted to achieve: detect a cross-state initiation.

Overall, this memory map organization delivers memory buffers in a *non anonymous* fashion, in which all the memory requested by an LP are guaranteed to be a sub-portion of the contiguous virtual memory address region reserved for that specific LP, if it is locally hosted. As for the case of a remote LP, a memory stock will be in any case initialized for it, but will be never used to serve memory requests.

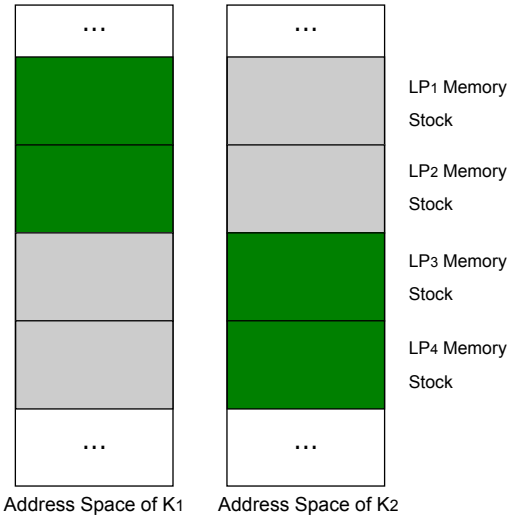

Figure 3.2: LP memory map organization

3.3 Kernel Level Support

Cross-state detection and management is ultimately supported thanks to the usage of ad-hoc operating system's kernel facilities, which are basically exploited in our custom *Loadable Kernel Module* (LKM). This module is mainly based on two kind of interactions: an *explicit* and an *implicit* one. The former is supported by a set of `ioctl` commands, to provide the worker threads a method to alert the above simulation kernel that a certain logical process is starting to process an event. The latter allows the operating system kernel to notify whenever a LP is accessing a remote LP's state.

3.3.1 Explicit Interaction

Upon loading, the LKM generates a special single-access device file in the path `/etc/ecs` that, at simulation startup, the simulation kernel opens in order to notify the module that the worker threads spawned by it are to be managed with a set of rules defined by the module. In particular, the simulation kernel instance tells the module what range of virtual addresses dedicate to a specific LP through the `SET_VM_RANGE ioctl` command. Since the remainder of this chapter is strongly related to x86_64 architectures virtual to physical address mapping, it is important to introduce it in order to better understand the rest of the proposed solution.

When dealing with low-level memory management, the virtual to physical address translation is achieved via a technique called *paging*, since a *page* is the smallest memory handable unit. This method is based on storing some data structures, called *page tables*, in order to constantly have a consistent saved mapping. This kind of architectures, in particular, support (at most) 4 levels of page tables, each one of them addressing, in a single entry, the lower level one, until the last one which is pointing to the physical address of the actual 4 KB page. The 4 tables have a capacity of 512 entries, being thus ultimately able to map up to 512^4 entries, which means in turn 256 TB of total addressable memory: the tables are called, in order from the highest level one to lowest, *Page Map Level 4* (PML4 which is equivalent to *Page Global Directory*, PGD), *Page Directory Pointer Table* (PDPT, equivalently *Page Upper Directory*, PUD), *Page Directory* (PD, equivalently *Page Middle Directory* PMD, whose entries are *Page Directory Entries*, PDE) and *Page Table* (PT, whose entries are the *Page Table Entries*, PTE).

A representation of this structure is depicted in figure 3.3. In particular, it can be noticed that every linear (virtual) address can be decomposed in five different parts, each of them useful to traverse the 4 levels of page tables, expressing the offset where to find the pointer, inside the corresponding page table,

to the lower one. The last displacement, of course, refers to the one within the physical page into which the memory access is falling. In order to know where to start the indirection procedure, the *Control Register 3* (CR3) stores the address of the first level page table, the PML4. Also, note that only 48 bits of the linear address, from the 64 available are used, letting the programmers have many free bits to use as they please still allowing to address a big amount of memory.

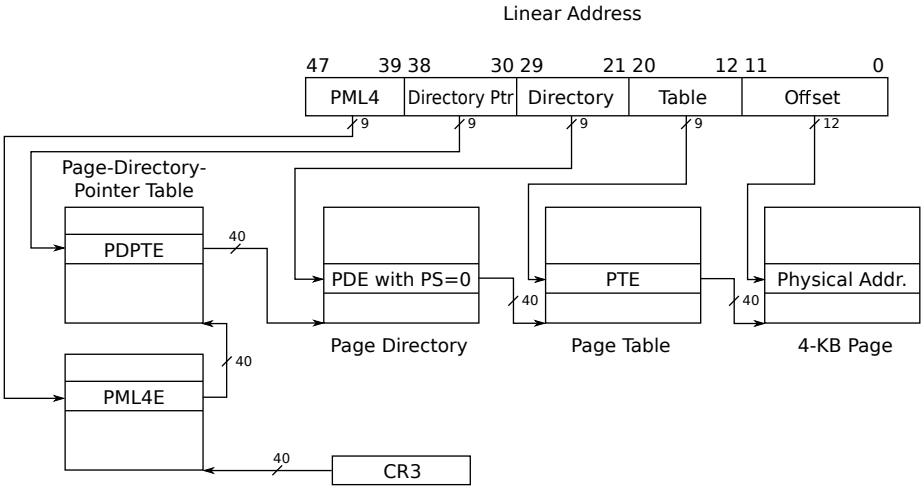

Figure 3.3: x86_64 paging scheme

The mapping described in the previous section, and depicted in figure 3.2 is realized basing on a specific rule: the 1 GB stock of memory dedicated to a LP is allocated in such a way that only one entry of the PDP table is involved. Therefore, any access to any physical page of a logical process can be associated with the actual LP simply via the PDP entry used in the virtual-to-physical translation. Indeed, memory stocks assignment results, in a very straightforward way, in calling the `SET_VM_RANGE ioctl` command passing as a payload the initial address of the first stock, that is the one dedicated to LP_0, and the overall number of LPs.

In order to detect whether an LP is accessing another one's memory, another technique is adopted. First of all, the worker threads inform the simulation kernel about what will be the next scheduled LP via the `SCHEDULE_ON_PGD ioctl` command. This command triggers the LKM logic that in turn essentially installs a copy of the current page table, namely a *sibling page table*, in the CR3 of the CPU core where the worker thread is currently running. This copy is performed by simply cloning both the PML4 related the virtual memory of the current process, retrievable accessing the `current` macro (more precisely, via `current->mm->pgd`) and the PDP tables that point to the actual LPs states. However, all those PDP entries but the one of the currently running LP are manually zeroed, in such a way that when an access towards a different LP's

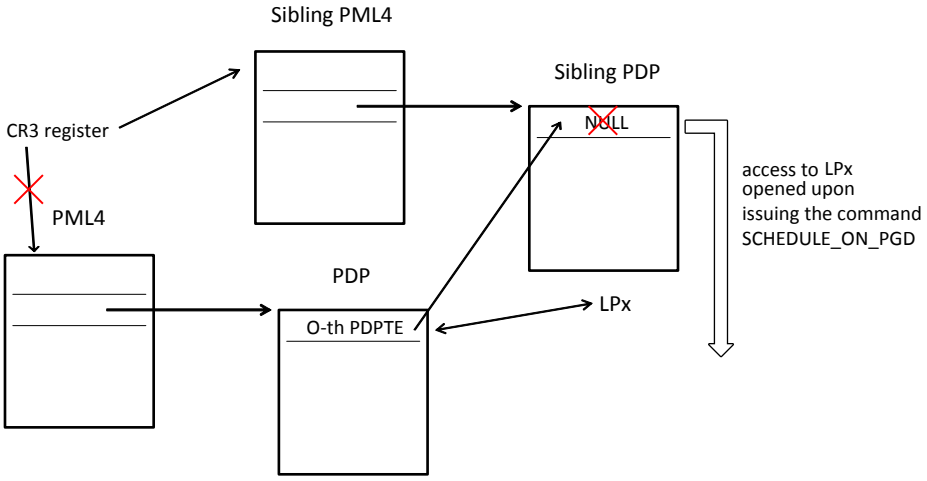

Figure 3.4: ECS schedule example.

state is tried, a memory fault is (automatically) generated. Regarding this last concept, note that when a memory stock is initialized it is important to actually *materialize* the entries of the whole chain of tables in order to really have them filled and to create consistent *sibling* pages. This is done by not only relying on `mmap` primitive, since it would leave the target memory in a empty-zero state (in fact, the kernel will finally materialize it only when strictly needed). Thus, what is done is that, beyond calling `mmap`, a null byte is written into the single virtual page of the stock.

Finally, it is important to highlight that the same result about LP cross-state accessing couldn't be reached exploiting more traditional facilities, such as functions like `mprotect`. In fact, this offers the user the possibility to block the access to a range of memory at the grain of the process, while the described approach allows multiple threads within the same process observe different memory accesses privileges, paying a negligible computational cost. Moreover, the coarse granularity of that kind of functions could lead to an erroneous situation while detecting ECS synchronization scenarios. An example of the sibling page table scheme is shown in figure 3.4.

3.3.2 Implicit Interaction

In order to make use of the sibling page tables structures, the simulation platform needs a technique to intercept and understand artificial memory faults generated by accesses falling in those cloned tables. Thus, the *Interrupt Descriptor Table* (IDT) is properly modified in such a way that the entry containing the pointer to the traditional `do_page_fault` kernel function, which manages memory faults, now points to an ad-hoc ECS fault handler. This handler logic follows

the traditional one enhancing it in order to cope with cross-state accesses.

The pseudocode of this handler is presented in Algorithm 4.

Algorithm 4 ECS Page Fault Kernel Handler

```
1: procedure FaultHandler(pt_regs* regs)
2:     if current → mm = NULL then                                  ▷ F1
3:         DoPageFault( )
4:         return
5:     end if
6:     if current → pid is not registered then                      ▷ F2
7:         DoPageFault( )
8:         return
9:     end if
10:    target ← ReadCR2( )
11:    if PML4(target) not in LP range then                         ▷ F3
12:        DoPageFault( )
13:        return
14:    else
15:        if PDP(target) = NULL then                               ▷ F4
16:            fault_type ← Major
17:        else
18:            if GetPteStickyBit(target) then                      ▷ F5
19:                fault_type ← Minor
20:                SetPresenceBit(target)
21:            else
22:                if ¬GetPresenceBit(target) then                  ▷ F6
23:                    DoPageFault( )
24:                    if GetPdeStickyBit(target) then
25:                        fault_type ← Minor                       ▷ F7
26:                        SetPageStickyFlag(target)
27:                    else
28:                        return
29:                    end if
30:                else                                             ▷ F8
31:                    fault_type ← AccessChange
32:                    SetPagePrivilege(target, WRITE)
33:                end if
34:            end if
35:        end if
36:    end if
37:    Switch to the original Page Table                            ▷ F9
38:    Copy to userspace fault information
39:    Push on userspace stack regs → ip
40:    regs → ip ← EcsHandler                                       ▷ F10
41: end procedure
```

First of all, when it is activated the ECS handler needs to understand if it is a fault not concerning cross-state synchronization, since in that case the traditional fault handler needs to be called. The cases in which this happens are:

- The handler is activated to solve a fault from kernel space (**F1**).

- The process who generated the fault is not registered, i.e. it didn't open the `/etc/dev` device file (**F2**).
- The targeted address is not falling inside a PML4 entry related to LP memory stocks (**F3**).

If one of the three cases above arise, then the traditional DoPageFault kernel facility is invoked.

Then, the algorithm checks what kind of access a LP is trying to perform towards other LPs. In particular, if the PDP entry evaluated basing on the target address is zeroed (**F4**), it means that the faulting LP is generating a cross-state access for the first time, falling in a scenario named *ECS Major Fault*. Indeed, the `UNSCHEDULE_ON_PGD ioctl` command explicitly clears PDP entries which are reserved for different LPs. In this case, we simply change the *Instruction Pointer* (IP) value to make it point to a userspace platform function called EcsHandler (**F10**). However, before giving back the control to platform, the original page tables view is re-installed restoring the old value of the CR3 (found at `current->mm->pgd)` and all the informations about the fault (such as fault type, fault memory target and address of the faulting instruction) are copied to userland in a per-thread buffer (**F9**). The execution flow is then resumed by simply pushing on userspace stack the original value of the IP.

The userspace EcsHandler starts a (potentially distributed) synchronization algorithm that will be later discussed. When synchronization is started with a different LP, the kernel module needs to understand which pages are accessed and in what mode, in order either fetch the requested content from the targeted LP or to update it. Thus, the `IOCTL_PROTECT_REMOTE_LP ioctl` command, which is in charge of protecting the memory in a way similar to the `mprotect` primitive, is invoked.

In order to overcome the aforementioned limit that the `mprotect` facility presents, the PTE structure, shown in figure 3.5, is exploited. In particular, starting from the PDP related to the LP the synchronization is targeted to, all the page table entries attainable from it are scanned. According to the operating system kernel default behavior, all the PTE entries which are not-null have the *presence bit* set, being them already materialized. The algorithm explicitly forces this bit to be unset, hence generating an additional artificial page fault the next time the page will be accessed. However, in order to understand whether the fault is artificial or not, along with the *presence bit*, also a *sticky bit*, the one at position 9, is set. From the picture above (figure 3.5) in fact we can notice that the bits in the interval [9,11] are not used (they're *available bits*), and thus can be manually set by developers to program a custom behavior not supported by the firmware. Also, when this operation is performed, the bit 11 of the associated PDE entry is set, in order to mark the whole stock as the one

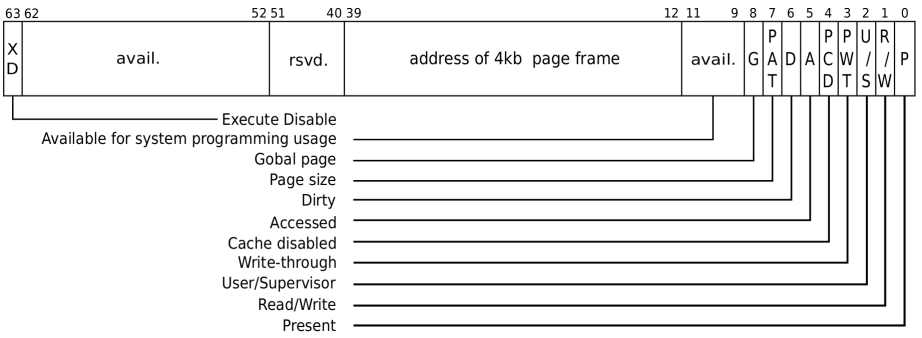

Figure 3.5: Page Table Entry (4 KB page).

of a remote LP. Eventually, the LP generating the ECS synchronization will be re-scheduled and it will generate an (artificial) page fault due to the access to the page whose PTE was manually modified. At this point, the algorithm checks whether the page was already materialized, condition that could arise if a previous execution of an ECS synchronization was put in place. This is done, at points **F5**, **F6** and **F7** by inspecting the value of the *sticky bit*. If the checked value is 1, then the *presence bit* is restored to 1, and the retrieval of the required pages is allowed to start by delivering to user-space the information that the happening scenario is the *ECS Minor Fault*. If, otherwise, the *sticky bit* is unset, the page needs to be materialized if and only if the *presence bit* is unset as well (**F6**), which is done by simply invoking the standard DoPageFault kernel handler. Finally, the algorithm determines whether the page fault being handled is related to local or remote pages, by inspecting the value of the bit 11 of the PDE entry. If this bit is set (**F7**), then we're in the case of an *ECS Minor Fault* and this is notified to the user-space ECS handler just after having set all the *sticky bits* of the PTE entries related to that PDE, in order to bring back the PTE table to a consistent state.

Moreover, another consideration about the check at line **F6** is to be done. In particular, this check covers another case related to the access mode to the page. When, in fact, a remote page is accessed in read mode, the possibility of accessing it in write mode is explicitly prevented by setting the bit at position 1 of the associated PTE entry via the `IOCTL_SET_PAGE_PRIVILEGE ioctl` command. This bit is called *read/write bit* and, when unset, generates a page fault if the targeted page is accessed in write mode. When this is the case, the bit is manually brought back to one (**F8**) enabling the possibility to update values on the page. Also, the user-space is notified about this change thanks to the delivery of an *ECS Access Change Fault* notification. This final note is important due to the optimizations about multiple ECS synchronizations changing the access mode discussed in the remainder of the chapter. As a final remark, note that the proposed scheme supports both the standard 4 KB page size and

large pages, namely 2 MB sized. The first case is the one described above, involving the whole 4 levels chain of page tables, while the latter only entails three levels: PML4/PDP/PDE. Indeed, the fault handler is able to understand what kind of pages are currently used, and set up the sibling page table's structure accordingly. Moreover, the `swapon/swapoff` facilities, natively supplied by the operating system kernel, were exploited in order to temporarily disallow asynchronous updates of the original page table chain issued by the `kswapd` swapping daemon.

3.3.3 The Distributed Synchronization Protocol

Whenever a cross-state request is detected, the involved LP will need to synchronize since, in order to update each others memory, the WCTs must be aligned not to generate inconsistency in both the two entities. This is achieved by control message passing among LPs, and the scenario we will refer to is depicted in figure 3.6. The synchronization starts when the system detects that an LP_x is trying to access a memory area owned by LP_y.

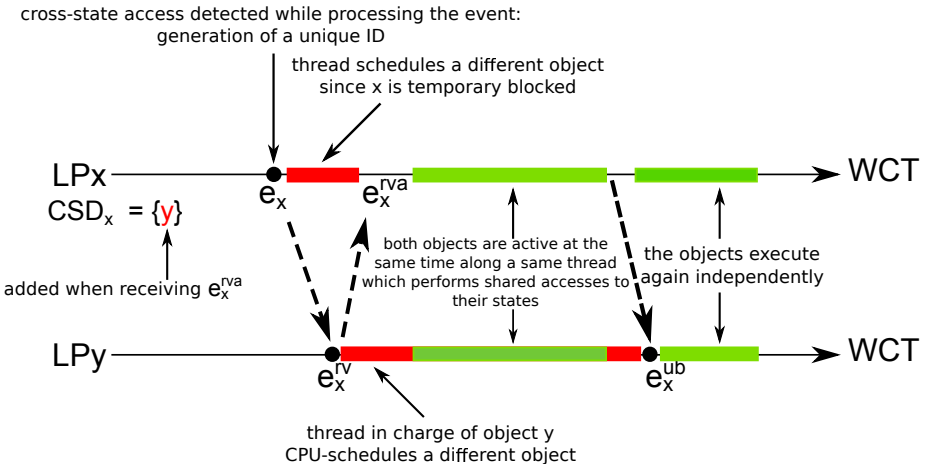

Figure 3.6: LP synchronization scheme

This triggers the *rendezvous* control message passing logic, which consists of the following steps:

1. LP_x sends a *rendezvous start* control message e_x^{rv} directed to LP_y and associated to a system-wide unique *rendezvous mark*.

2. LP_x execution is suspended (condition reached thanks to the above described ULT technique) and its state is switched to the Wait For Synch one.

3. Eventually LP_y will *i*) receive the *rendezvous start* control message and incorporate it inside its event queue, *ii*) reach the execution time of this event either due to forward execution or due to a rollback procedure if e_x^{rv} represented for this LP a *straggler* message. LP_y thus generates a *rendezvous ack* control message e_y^{rva} carrying the same *rendezvous mark* of e_x^{rv} in order to notify LP_x the correct delivery of the ECS request.

4. LP_y is put in the Wait For Unblock state, which is a blocked state, in order not to execute events that may change the memory view LP_x is accessing.

5. Once e_y^{rva} is delivered at LP_x, the system switches its state to Ready For Synch, allowing this LP to be eventually reactivated. LP_y is also inserted into the *Cross State Dependency* table of LP_x. This is an important data structure since it is passed as an argument to the `SCHEDULE_ON_PGD ioctl` command, determining thus what PDP entries should be opened for access in order to build the sibling page table scheme for the core running the worker thread.

6. The two LPs are now synchronized: they share the same logical virtual time and it is safe now for LP_x to access LP_y's state. If the latter LP is remote, then the accesses will generate additional control messages, which constitute a slight improvement, presented later in this chapter, to the adopted approach.

7. When LP_x finishes the operations on the state of the other LP, i.e. it completes the execution of the currently scheduled event, it terminates also the synchronization by sending to LP_y a *rendezvous unblock* control message e_x^{ub}.

8. The states of both the LPs are restored to Ready, and execution can proceed normally.

Note that this procedure can be iterated multiple times, in such a way that within the execution of a single event, LP_x is able to synchronize with more than one LP at a time. In this scenario, the same *rendezvous mark* is used. This is a core aspect since in this way, whenever any of the involved LPs needs to rollback, the synchronization is aborted and all the LPs involved in a synchronization with it would rollback as well.

3.3.4 Userspace ECS Management

Algorithm 5 shows the logic triggered upon LKM runtime notification of the cross-state synchronization request issued by a logical process. This algorithm handles both the local and remote ECS synchronization request cases. As described in the above section, the ECS Major Fault (**H1**) is related to the initiation of the scheme which follows the (distributed) described protocol. First of all, a system-wide unique *rendezvous mark* is generated, and the *rendezvous start* control message is sent to the target LP that corresponds to the one keeping the portion of simulation state which is currently accessed by the executed event (as stated in section 3.3.1). In fact, the ID of the target LP is associated strictly to only one entry of the PDP table related to the faulting memory address, and it is delivered to the handler by the LKM.

Algorithm 5 Userspace ECS Handler

```
1: procedure EcsHandler(type, info)
2:     if type = Major then                                   ▷ H1
3:         ECS_mark ← generate_mark( )
4:         Send(RENDEZVOUS, info.targetLP, currentLVT)
5:         LP_state ← WAIT_FOR_SYNCH
6:         CSD ← CSD ∪ {info.targetLP}
7:         Deschedule( )
8:     else if type = Minor then                              ▷ H2
9:         disasm ← Disassemble(info.rip)
10:        write_mode ← disasm.write
11:        page_addr ← BaseAddr(info.target)
12:        pages ← PgCount(info.target, disasm.span)
13:        if write_mode then
14:            AddToWriteList(page_addr, pages)
15:        else
16:            AddToReadList(page_addr, pages)
17:        end if
18:        Send(PAGE_LEASE, info.targetLP, currentLVT)
19:        LP_state ← WAIT_FOR_PAGE
20:        Deschedule( )
21:    else if type = AccessChange then                       ▷ H3
22:        page_addr ← BaseAddr(info.target)
23:        AddToWriteList(page_addr, 1)
24:    end if
25: end procedure
```

Therefore, the running LP's state is set to Wait For Synch and the target LP identifier is inserted into the *Cross State Dependency* set (CSD) which is ultimately represented by the aforementioned Cross State Dependency table. The currently running LP is then descheduled, exploiting the ULT facilities mentioned above, in order to block its execution until a *rendezvous ack* control message is received.

If the type of the ECS fault is minor (**H2**), it means that an access to the

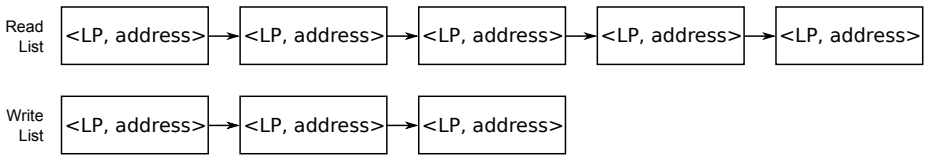

Figure 3.7: Page touch lists

state of a remote LP was tried. Therefore, it is needed to discriminate what kind of request was performed, namely a read or a write operation. To cope with this issue, the algorithm relies on a in-place dynamic disassembly [1] of the instruction, passed within the *info* argument by the LKM, which can be found in the model's address space by inspecting the address which caused the memory fault. This disassembler provides, along with the just mentioned one, several important informations related to the faulting instruction. Indeed, for instance, it gives the possibility to know whether the fault was generated due to a read or a write operation, and the amount of memory that the operation involved. This latter information is crucial since it is used, together with the memory address where the fault occurred, to calculate the base address of the first (remote) page and the number of pages which are to be transferred to the target LP. This information is in fact piggybacked by an additional control message sent, before entering the Wait For Page state, towards the target LP, and it takes the name of *page lease* since the requesting LP is acquiring a page, logically controlling it for a specific amount of time. This means that it temporarily owns a master copy of the data stored in the leased pages, which represent a portion of state. Those pages can be safely installed in the local LP address space thanks to the non-overlapping organization presented in figure 3.2. Since the number of the leased pages can vary and possibly increase, two *touch lists* are maintained: one is dedicated to the pages requested in read mode (the *read list*), and the other for the one accessed in write mode (the *write list*). The entries of those lists not only keep the addresses of the interested pages, but also the real LP owning that page. In fact, a LP is allowed to synchronize with any number of LPs, so the ownership of the page must be remembered. Technically each node of the list is associated to a single PTE entry, and a 512 bits bitmask (one bit for each entry of the PTE) is stored with the scope of determining whether the corresponding page was gained or not.

Accessing a page in read mode would mean, for an LP, to already have acquired a lease on it and to already have its content installed in the local address space. However, accessing the same page in write mode would generate a page fault since the underlying operating system granted the access to it only

[1] This disassembler is a part of a bigger software called Hijacker, developed by the HPDCS research group and foundable at https://github.com/HPDCS/hijacker

in read mode. Nonetheless, this kind of fault can be managed locally by the LKM and the user-space ECS handler. In fact, upon this kind of request the LKM updates the access rights (see **F8** in Algorithm 4) before giving back the control to the users-pace handler. Thus, the latter performs the only task of switching the touch list of that page, from the read to the write one.

Furthermore, the role of the write list is not limited only to store the leased pages. When synchronization is about to end, the LP who started the ECS needs to send a *rendezvous unblock* control message in order to allow all the synced LPs to restart execution. Hence, those LPs need to be informed about what pages were updated and what is the new content they need to install in their local address space. This is performed by augmenting the unblock message by adding as payload all the pages for which a lease in write mode has been acquired during event execution. At the recipients side, the memory view is updated immediately before restarting the execution of the LP, in order to reconciliate the remote states and to share a consistent state with respect to the logic of the just executed handler.

Chapter 4

Experimental Results

We remind the reader that the proposed solution was integrated within ROOT-Sim, a general purpose parallel/distributed Open Source C/MPI based optimistic simulator, which exposes the explained facilities transparently to the model developer. In order to evaluate interesting experimental results, we needed to choose a well suited simulation model as a benchmark application. To this end, a multi-robot exploration and mapping simulation model, following the results of [FKK+06] was selected. In particular, the main actors of this models are robots whose scope is to explore the unknown space they're inserted in, by means of the sensors they're equipped of (lasers, cameras, etc.). Specifically, the area to explore is a squared region divided in hexagonal cells. The robots exploit their sensors in order to build a map during the exploration of the world in order to accomplish their task in a more efficient way. Moreover, every robot has the ability to determine an *exploration frontier*, which is defined as the closest unexplored area reachable from the current position. Hence, whenever a robot needs to figure out where to proceed in the exploration, it relies on this concept, computes the fastest way to reach the destination point and continues the exploration. The number of cells and exploring robots can be determined at application startup and they perform their task in an independent way as long as they coincidentally notice the presence of another of them in the nearby. Whenever this happens, we say that a robot enters a *proximity area* and the associated actions performed by the involved robots in this circumstance are the following:

1. they estimate their mutual position collecting the data retrieved by their sensors. Recall that in this scenario robots are only in a *proximity area*, which means that they are not necessarily positioned in adjacent cells.

2. they generate a *rendezvous* (i.e. appointment) point in the map where to meet again in order to estimate the goodness of their current estimated mutual position

3. if they success in meeting again at the *rendezvous* point, they share each other the acquired data in order to shorten the exploration time and increase the precision of the future movements that they will opt for.

Furthermore, if the robots manage to meet again at the *rendezvous point*, they start to cooperate constituting a so called *cluster*, exploiting teamwork to accomplish their task. This collaboration is composed by two main sub-tasks: *i*) collectively establish their next target exploration areas, basing on the *cost* and *utility* functions also described in [FKK+06] and *ii*) they try to define a prediction on the position of the robots not forming the cluster they're part of. In particular, the robot for which is more advantageous to perform the latter operation (namely, the one for which the *cost* and *utility* functions returned more suitable values with respect to the target position) is demanded to actually complete it. If the guessed hypothesis is verified, the steps described above are carried out again, so that the cluster is possibly enlarged and simulation speeded up. Since robots are likely to exchange possibly considerable amounts of data (namely, explored maps, guesses, etc.), this kind of model is well suited to test the behavior and the performance of the provided distributed ECS solution.

In particular, we ran the experiments on a HP ProLiant server mounting 8 AMD Opteron 6376 CPUs, running at 2.6GHz each and equipped with 100 GB of RAM. Each CPU includes four cores, for an overall total of 32 physical cores. Furthermore, the actual simulation was executed on a cluster of virtualized nodes made up by 10 Virtual Machines (VMs) hosted by VMware Workstation hypervisor, version 10.0.4 build-2249910. On top of every virtual machine, Debian 6.0.7 was installed, thus relying on Linux Kernel 3.16.0-4. The cluster was created in such a way to reproduce a mid-range computer set, since every VM rely on 8 GB of memory and 2 virtual cores. Finally, the model was configured to have a map formed by 484 hexagonal cells, which are explored by 4 different robots. In order to get a good comparison between the traditional communication pattern and our optimized ECS solution, we ran tests in two different approaches:

- **ECS**: exploiting the actual transparent distributed synchronization protocol described in this thesis. It is based then on memory pointers while *i*) exchanging data between robots and *ii*) registering one robot into a cell (note that upon initialization, each cell stores its state in a shared array which is mirrored in all computing nodes).

- **No-ECS**: basing on classical message passing as an alternative to implement the same logic described in the previous point. For instance, when robots are required to exchange each other the fraction of the map they explored, data is simply inserted into an event's payload and it is piggybacked to the recipient. This clearly represents a drawback since the model design complexity is increased, a problem that was directly tackled in this thesis.

In the chart of figure 4.1 the simulation throughput, when changing the number of computing nodes, is drawn, showing two main different behaviors. In particular, when the number of nodes does not overcome three we can notice that the presented solution provides an improved performance compared to the traditional technique. This is mainly caused by the fact that while running simulation events, the involved robots are accessing a non-minimal portion of each others data both in read and in write mode. However, this is also the case where the probability that LPs reside on the same node is very high (approximately 50%), consequently increasing the likelihood of accessing a simulation state which is already present in the machine they're running on. This situation is clearly a case in which our solution is disadvantaged, since the exchanged data typically occupies around three pages, which can be easily and better optimized transferred with the traditional method (event data marshaling/unmarshaling). Nonetheless, when the number of nodes increases, while overall number of LPs is kept fixed, thus incrementing the degree of parallelism, the overhead introduced by our solution only increases up to 30%. Keeping in mind that while synchronizing logical processes block their execution for a non-minimal amount of WCT, exchange a high number of control messages and that the low number of LPs drastically increases the chance of delivery of *straggler* messages by LPs involved in a synchronization (thus entailing rollback operations and forcing ECS protocol to retry multiple times), we can state that the evaluated overhead value mentioned above represents a good result. Indeed, despite the adverse scenario, the protocol is able to keep this overhead quite reduced.

Finally, in figure 4.2 we show the speedup, with respect to the number of nodes, when running the same model in purely sequential way and executing making use of an efficient Calendar Queue (a data structure presented in [Bro88]).

By the results, we can observe that the introduced speedup is non-minimal, showing that the experimental assessment was conducted comparing competitive parallel runs. Still, although the low number of LPs is limiting the speedup in our ECS solution, that speedup is always increasing, leading the reader to expect that in the best-case high-load, inter-kernel communication intensive widely-distributed scenario it can overtake the No-ECS execution case.

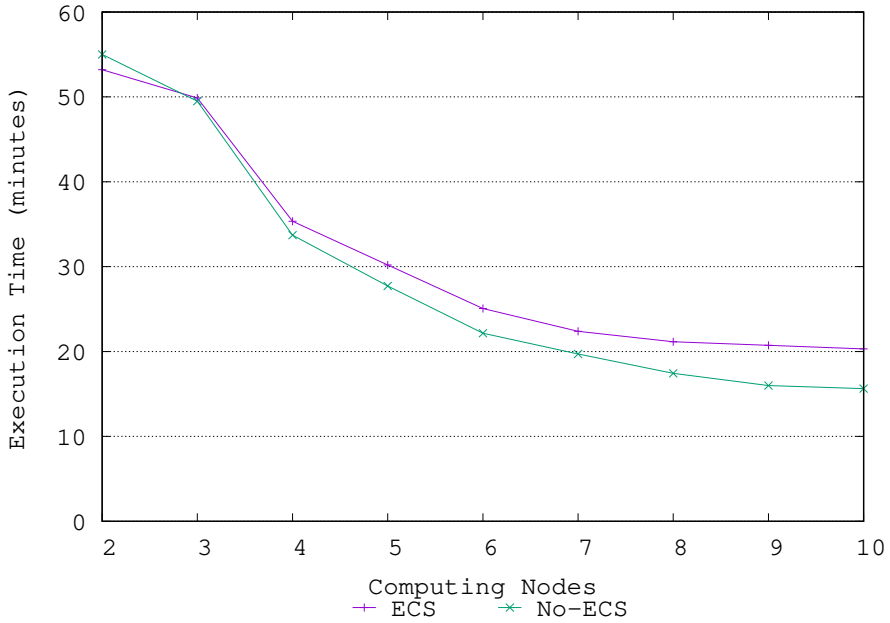

Figure 4.1: Simulation throughput

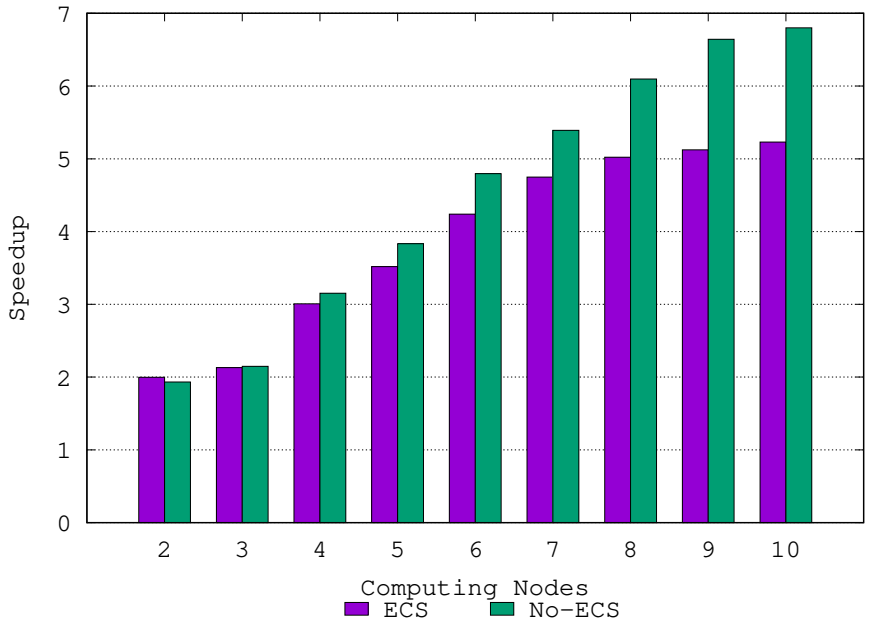

Figure 4.2: Speedup with respect to sequential simulation

Chapter 5

Conclusions and Future Work

In this thesis we presented an innovative approach to directly tackle the problem of distributed cross-state synchronization in PDES. In particular, the described architecture allows LPs to rely on sequential-style programming paradigm to access in read or write mode the simulation state of any local or remote LP lying in the distributed simulation system.

This thesis specifically aimed at significantly simplifying the programming model, giving the model designer a high amount of flexibility in developing its system being it not forced to care about synchronization issues. Overall, the designer is asked to develop using a fully sequential style paradigm, demanding the underlying platform to care about simulation concerns.

Moreover, this solution was meant to be transparently deployed on generic and general purpose distributed environments. Indeed, the tests were executed on a cluster of virtualized computing nodes, which clearly denotes the universality of the approach. The discussed work is only constrained by the usage of Linux 3.16.0-4 kernel facilities, running on top of x86-64 architectures. Given the wide spread exploit of those two requirements, they ultimately don't represent an obstacle to the transparent deployment of this system.

The worst-case scenario used to collect the previously presented experimental assessments shows that our solution can be considered well viable from a performance point of view, being the introduced overhead almost negligible under unfortunate circumstances.

This work, that finally resulted in the [PPQ18] article submitted to the *International Symposium on Cluster, Cloud, and Grid Computing* (CCGRID) 2018, opened a path in this research field, giving us many ideas about possible future works. Specifically, since this work aimed at generating *page leases* whenever

an LP accesses a remote LP's state memory portion, our idea is to tune the approach estimating what portions of memory an LP is going to access. In fact, thanks to the concept of *temporal locality* we can understand if a LP is requesting a certain portion of memory at a high frequency, giving the LP the right of owning it for longer periods. Also, exploiting the *spatial locality* principle, we can make guesses about pages which are likely to be requested by an LP, in order to obtain the *lease* of a set of pages that may be useful in the future. We believe that both this two techniques can drastically reduce the overhead generated by the synchronization protocol, since in both cases the number of times it would be started should be significantly cut off. Therefore, less messages would be exchanged and the overall per-LP elapsed time in a blocked state would be radically lowered down, consequently leading to a gain in performance.

Bibliography

[Bel90] Steven Bellenot. Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 122–127, January 1990.

[Bel92] Steven Bellenot. State skipping performance with the Time Warp operating system. In *Proceedings of the 6 th Workshop on Parallel and Distributed Simulation (PADS '92)*, pages 53–64, 1992.

[BP29] Robert Baden Powell. How the acorn became an oak tree. *Boy's life*, XIX(2):9, 1929.

[Bro88] Randy Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[CLY+11] Li-li Chen, Ya-Shuai Lü, Yi-Ping Yao, Shao-Liang Peng, and Ling-Da Wu. A well-balanced time warp system on multi-core environments. In *25th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation, PADS 2011, Nice, France, June 14-17, 2011*, pages 1–9, 2011.

[CM79] K. Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[CPF99a] Christopher D. Carothers, Kalyan S. Perumalla, and Richard Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, 1999.

[CPF99b] Christopher D. Carothers, Kalyan S. Perumalla, and Richard Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, July 1999.

[DF97] Samir R. Das and Richard M. Fujimoto. Adaptive memory management and optimism control in time warp. *ACM Trans. Model. Comput. Simul.*, 7(2):239–271, 1997.

[DFP+94] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. GTW: a time warp system for shared memory multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pages 1332–1339. Society for Computer Simulation International, 1994.

[FD97] 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, Lockenhaus, Austria, June 10-13, 1997*, pages 82–89, 1997.

[FKK+06] 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.

[Fuj89a] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, 1989.

[Fuj89b] Richard M. Fujimoto. The virtual time machine. In *Proceedings of the fisrt annual ACM symposium on Parallel algorithms and architectures*, pages 199–208. ACM Press, 1989.

[Fuj90] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proceedings of the Multiconf. on Distributed Simulation*, pages 23–28. Society for Computer Simulation, January 1990.

[Fuj93] Richard M. Fujimoto. Parallel and distributed discrete event simulation: algorithms and applications. In *WSC '93: Proceedings of the 25th conference on Winter simulation*, pages 106–114. ACM Press, 1993.

[GF91] Kaushik Ghosh and Richard Fujimoto. Parallel discrete event simulation using space-time memory. In *Proceedings of the International Conference on Parallel Processing, ICPP '91, Austin, Texas, USA, August 1991. Volume III: Algorithms and Applications.*, pages 201–208, 1991.

[Gos03] Abhijit Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[Jef85] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, July 1985.

[Jef90] David R. Jefferson. Virtual Time II: storage management in conservative and optimistic systems. In *Proceedings of the 9th annual ACM symposium on Principles of distributed computing*, pages 75–89. ACM, 1990.

[LL89] Yi-Bing Lin and Ed D. Lazowska. Determining the global virtual time in a distributed simulation. Technical report, University of Washington Department of Computer Science and Engineering, January 1989.

[McK04] Sally a. McKee. Reflections on the memory wall. *Proceedings of the first conference on computing frontiers on Computing frontiers - CF'04*, page 162, 2004.

[MNPQ16] Nazzareno Marziale, Francesco Nobilia, Alessandro Pellegrini, and Francesco Quaglia. Granular time warp objects. In *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS 2016, Banff, Alberta, Canada, May 15-18, 2016*, pages 57–68, 2016.

[PDQ11] Sebastiano Peluso, Diego Didona, and Francesco Quaglia. Application Transparent Migration of Simulation Objects with Generic Memory Layout. In *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*, pages 169–177. IEEE Computer Society, 2011.

[PPQ18] Matteo Principe, Alessandro Pellegrini, and Francesco Quaglia. Transparent Deploy of Sequentially-Coded PDES Models on Distributed-Memory Systems. In *EEE/ACM International Symposium on Cluster, Cloud, and Grid Computing CCGRID*, 2018.

[PPQV16] Alessandro Pellegrini, Sebastiano Peluso, Francesco Quaglia, and Roberto Vitali. Transparent speculative parallelization of discrete event simulation applications using global variables. *International Journal of Parallel Programming*, 44(6):1200–1247, 2016.

[PQ14a] Alessandro Pellegrini and Francesco Quaglia. The ROme OpTimistic Simulator: A Tutorial. In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, PADABS, pages 501–512. LNCS, Springer-Verlag, Aachen, Germany, 2014.

[PQ14b] Alessandro Pellegrini and Francesco Quaglia. Transparent Multi-Core Speculative Parallelization of DES Models with Event and Cross-State Dependencies. In *Proceedings of the 2014 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pages 105–116. ACM, 2014.

[PQ17] Alessandro Pellegrini and Francesco Quaglia. A fine-grain time-sharing time warp system. *ACM Trans. Model. Comput. Simul.*, 27(2):10:1–10:25, 2017.

[PVQ09] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Didymelor: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53, Washington, DC, USA, 2009. IEEE Computer Society.

[PVQ15] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Trans. Parallel Distrib. Syst.*, 26(6):1560–1569, 2015.

[RAT93] Hassan Rajaei, Rassul Ayani, and Lars-Erik Thorelli. The local time warp approach to parallel simulation. *SIGSIM Simul. Dig.*, 23(1):119–126, 1993.

[Rey88] Paul F. Reynolds, Jr. A spectrum of options for parallel simulation. In *Proc. of 1988 Winter Simulation Conference*, pages 325–332. Society for Computer Simulation, December 1988.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[TPQ+17] Tommaso Tocci, Alessandro Pellegrini, Francesco Quaglia, Josep Casanovas-Garcia, and Toyotaro Suzumura. ORCHESTRA: an asynchronous wait-free distributed GVT algorithm. In *21st IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2017, Rome, Italy, October 18-20, 2017*, pages 51–58, 2017.

[TQ08] Roberto Toccaceli and Francesco Quaglia. Dymelor: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172, Washington, DC, USA, 2008. IEEE Computer Society.

Acknowledgments

First of all, I would like to thank my advisors proff. Bruno Ciciani and Francesco Quaglia, who gave me the possibility to get into this field of computer science, which is as hard as challenging, and offered their knowledge and competences to introduce me in the world of scientific research. I give my best thanks to Alessandro Pellegrini, a true lifelong friend with whom I shared the craziest moments in the development of this thesis: without his help, his sustain and our teamwork this thesis wouldn't have been possible to realize. I would like also to thank all the members of the HPDCS (and, eventually, *Lockless* S.r.l) research team, a group made by kind, nice and skilled people who improved my knowledge and, primarily, made me feel at home (overall, I used your desks more than my own one at my place!).

Then, I cannot forget my family. My father, my mother and my sisters always believed in me and gave me the (financial) possibility to reach this result concurring at my realization as a man. The support they always showed me in the countless times I was unbearable, and the love they gave me in all this long period of studies were the light that allowed me to overcome the darkest times I needed to face in these years. Also, my sisters Chiara and Elena taught me that persisting in our choices is something to believe in, even if it implies not to live under the same roof. Thank you for understanding my sometimes hard personality, and to give me smiles and laughs in the moments i needed.

Thanks to Tommaso, Djordje and Lorenzo, my best university colleagues and friends with whom I spent days and years in front of a screen, and without whom it would have been almost impossible to pass all the exams and to believe that we could achieve this great success.

Finally, thanks to all my friends from the Scouts family. Thanks to all the scout chiefs, that went and come, that contributed and still contribute in my personal fulfillment. Without you, Matteo wouldn't be the person he is, now. Baden Powell (founder of Scouts movement) used to say *the acorn* [...] *had grown a seeding plant* (cf. [BP29]): I may be still a little tree, but if I am, it is mostly thanks to you.

This work was entirely developed using open source software, such as TEX and LATEX, I would like to thank all the people that offered me those free-to-use tools.

The master thesis opened a new chapter in my life: i am really grateful to all the people who stayed with me along this path... but the best is yet to come!