Simulation is an attractive and well-consolidated methodology to study real-world phenomena. In particular, Parallel Discrete Event Simulation (PDES) is a paradigm which has been extensively used (because of its modular and simple way of specifying simulation models) and has been proven as very effective in a wide set of fields, including physics, biology and business-oriented processes (such as financial prediction or optimized system-configuration selection). A core aspect of PDES platforms is that they allow the exploitation of multiple computing units in order to give rise to a parallel execution of the simulation model, which can provide significant reduction of the time requested for delivering simulation outputs to end-users or applications. Specifically, PDES has been shown to provide (large) speedups when compared to classical simulation paradigms where the execution of the simulation model takes place in a sequential fashion. This enables the wide usage of simulation in contexts where timeliness in the production of the simulation results plays a fundamental role, such as when employing simulation technology in time-critical decision processes.

The PDES paradigm dates back to the 80's and was originally thought as of a means for exploiting computing systems formed by clusters of machines relying on single-core CPU technology. On the other hand, more recent technological trends lead to the proliferation and large diffusion of multi-core hardware, where multiple computing units are hosted on the same machine and share several hardware resources, such as main memory. This unavoidably lead to the need for rethinking the organization of PDES platforms in order to make them perfectly suited for exploiting the computing power offered by modern multi/many-core machines.

In this chapter we discuss some key aspects related to the reorganization process of these platforms and present in detail a recent literature approach exactly tackling this issue. The presentation is also targeted at showing how the approach, which is based on the symmetric

multi-threading software-programming paradigm, can be suited for a change in the perspective on how to exploit computing resources for PDES applications in a balanced and effective manner. This is achieved via an innovative load sharing paradigm suited for PDES systems run on top of multi-core machines.

The chapter also considers a case study in the context of optimistically synchronized PDES platforms, which are based on speculative processing schemes and achieve causal consistency of the parallel run by means of rollback/recovery techniques. The case study reports the main outcomes of an experimental assessment of the suitability of the revised organization of PDES systems and of the load sharing technique.

The remainder of this chapter is organized as follows. We initially provide background information in relation to the PDES paradigm, including some examples of PDES-compliant models, which may help the reader to fully understand the paradigm potential. Then we enter details related to the multi-core technological trend, also discussing the motivations for its adoption. After we outline the main challenges and opportunities related to the organization of modern PDES platforms to be run on top of multi-core machines. Successively, we provide an overview of the aforementioned recent proposal particularly aimed at achieving balanced exploitation of the available computing resources when running PDES models on multi-core systems. Finally, we present the case study on the design and implementation of an instance of PDES platform tailored for multi-core machines, based on the optimistic synchronization paradigm, and report some results related to its run-time behavior.

Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) [1] is considered as a de-facto standard for developing simulation systems featuring high performance executions of complex and generally-

applicable simulation models. In particular, PDES has been successfully exploited in differentiated contexts, including – but not limiting to – symbiotic systems, or simulation-based (time-critical) decision making.

It is evident that the possibility to rely on a highly-optimized PDES framework, able to exploit (large-scale) parallel/distributed computing platforms, provides various benefits:

- *Problem feasibility* is broadened: very complex/large models, demanding large/huge CPU and/or memory resources, become tractable;
- *Simulation accuracy* is maximized: the simulation-model writer is provided with the ability to study complex phenomena with an always increasing level of detail, enabling researchers from different fields to enhance the quality of simulation results;
- *Timeliness of results* can be ensured: in this way, simulation results can be effectively used in scenarios where, e.g., the outcome of simulations can drive a real/physical system (with proper timing constraints), in order to take the best decision, given the run-time variations of the surrounding environment.

The basic idea underlying PDES is to partition the simulation model into several distinct *simulation objects*, which are the core of the simulation process from the model writer's point of view. In fact, each object represents a portion of the real world being simulated, the evolution of which is described by object state transitions, driven by a set of logical/mathematical properties. In order to represent real-world interactions, simulation objects can communicate with each other, by exchanging pieces of information in the form of *events*.

From a technical point of view, simulation objects are handled by *Logical Processes* (LP), which undertake the concurrent execution of simulation events. Traditionally, a PDES run entails N concurrent LPs, uniquely identified by a numerical code in the range $[0, N-1]$, and the overall

simulation model keeps track of the evolution of the simulated world by relying on a global simulation state, which is partitioned into various LPs' private and disjoint simulation states. In particular, if we denote with S the global simulation state and with S_i the LPs' private simulation states, two properties hold:

$$S = \bigcup_{i=0}^{N-1} S_i \qquad (1)$$

$$S_i \cap S_j = \emptyset, \forall i, j \in [0, N-1] \qquad (2)$$

This means that, in order for two different LPs to interact, *events* must be exchanged in the form of explicit *message passing*, as it is not possible for a LP to update/modify any other LP's private simulation state.

In PDES, simulation events are *timestamped* and their execution is *impulsive*, meaning that there is no notion of time evolution during an event processing. The current simulation time at each individual LP is known as *Local Virtual Time* (LVT), and can be expressed in any measure unit (i.e., one LVT unit can represent seconds, hours, or even years, depending on the actual simulation model). This notion of time is opposed to the *Wall-Clock Time* (WCT), which is the actual notion of time which we – as human beings – are used to. Therefore, in one WCT unit, the LVT advancement can be of one or several units, depending on the actual complexity of the simulation model and on the efficiency of the simulation run.

During the execution of an event, other events can be generated – destined to any simulation object in the system –associated with a timestamp value greater than or equal to the one of the in-execution event. This means that during the execution of the event E_x associated with the timestamp T_x, a new event E_y associated with timestamp T_y can be generated and sent to any

simulation object, ensuring that $T_y \geq T_x$. Therefore, the simulation execution evolves according to a causality pattern where the present cannot affect the past.

The execution of a PDES run occurs into two modes, namely *application-level mode* and *kernel-level mode*. The former execution mode is associated with actual simulation events processing, i.e. the control flow is passed to some LP which schedules the involved simulation object. On the other hand, the latter entails all the procedures carried on by the simulation run-time environment to support the actual event execution (e.g., managing the LPs' queues, or sending messages to remote kernel instances). Tasks executed while in kernel mode are usually referred to as *housekeeping* operations, and as the reader can imagine, enhancing the overall performance of a PDES execution involves minimizing the time spent in housekeeping (i.e., maximizing the time spent in event processing). Overall, LPs deliver simulation events to the hosted simulation objects via the invocation of proper event handlers. On the other hand, simulation-kernel instances take care of dispatching event-processing activities across the various LPs, and of managing inter-LP communication. In particular, they handle the LPs' *event queues*, by reflecting the updates associated with incoming messages, and determine the best LP to be dispatched on a given computing unit in order to optimize specific execution metrics. A schematization of the traditional organization of a PDES platform is shown in Figure 1.

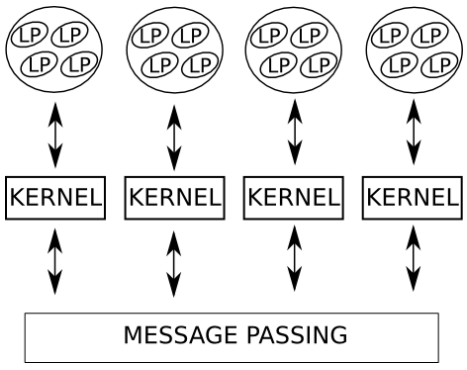

Figure 1 – Traditional Organization of Parallel Discrete Event Simulation Platforms

As depicted, LPs are hosted by different instances of the simulation kernel, whose interactions are carried out via message passing. As hinted, this type of organization has been originally devised in the 80's, in order to exploit computing platforms based on clusters of commodity machines equipped with a single processing unit. This organization led to consider a tight 1:1 mapping between simulation kernel instances and processing units as a de-facto standard for the design/development of PDES platforms.

PDES-compliant simulation models

As mentioned before, a simulation model compliant to the PDES paradigm is structured in simulation objects which have disjoint simulation states. We want to provide here the reader with the description of two sample simulation models conforming to the PDES paradigm. In no way we mean this dissertation to be complete and thorough, yet we provide them for the sake of clearness, hoping they will eliminate residual doubts on the PDES paradigm, if any. Additionally, we will use these models in the final part of the chapter, to empirically show the benefits which derive by the forthcoming dissertations.

Personal Communication Service (PCS)

PCS is a mobile phone simulation model [6] targeted at simulating the evolution of a wireless communication system adhering to GSM technology, where communication channels are modeled in a high fidelity fashion via explicit simulation of power regulation/usage and interference/fading phenomena on the basis of the current state of the corresponding wireless cell. The power regulation model is implemented according to the results in [28].

In PCS, one simulation object models the evolution of a single wireless cell, and any LP is in charge of driving the execution of simulation events occurring in a single cell, namely at a single simulation object. The state of each simulation object is essentially a set of lists of records,

keeping track of the current state of the wireless channels managed by the cell. Particularly, a channel can be either busy or free. In the former case, it means that some wireless communication is taking place via that channel, for which the state of the simulation object keeps track of the current value of the transmission power and of the current signal-to-interference ratio. It is clear that if no active wireless communication is handed-off between adjacent cells, then each simulation object evolves along simulation time in a fully autonomous manner, given that any event occurring at that object (such as the setup of a call involving a mobile device within that cell) will give rise to future events only destined to the same object (such as the completion event for the previously mentioned call). This generates a so-called embarrassingly parallel PDES run, where no coupling at all ever takes place between the different objects included in the model. The overall scenario looks like one where each individual simulation object can be seen as a separate simulation model and the PDES platform would be in practice running multiple separate models concurrently. On the other hand, when simulating realistic settings mobile devices can switch cells, which may give rise to hand-off of active calls. Hence, some level of coupling between pairs of simulation objects can arise, given that the simulation will evolve by generating some *leave* event of the ongoing call on the source cell and some *income* event for the ongoing call on the destination cell. This is the typical case to be faced by PDES platforms, which needs to be optimized in terms of run time dynamics. In this scenario, the PDES platform is in fact solving a larger model (e.g., it is simulating the evolution of a large coverage area of the GSM system) by concurrently simulating the evolution of its individual, interacting components, according to a spatial decomposition of the simulated system into individual wireless cells. We note that the optimization of the run-time dynamics does not only involve optimizing the actual message exchange across the different simulation objects when

cross scheduling of events takes place. Rather, several other aspects are involved, among which the one associated with the actual computing demand for the simulation of individual events at the different cells. More in detail, when the setup of a call is simulated, the CPU time required for determining the initial power assignment for the call depends on the current number of active channels within the cell, and on the current power usage on these channels. The more channels are busy, the higher is the CPU cost for determining the minimum power to be assigned to the newly started call in order to meet some signal-to-noise ratio (according to the results in [28]). Hence, the different simulation objects may exhibit different CPU requirements for simulating their events depending on the actual workload they sustain (e.g., in terms of frequency of call arrival for mobile hosted by the corresponding cell), which may be different for objects simulating cells in different spatial regions. How to exploit the overall computing power in order to achieve balanced advancement of the simulation time across the interacting objects with different CPU requirements for the simulation of their events is another central issue to be dealt with by the PDES run-time environment.

Traffic

The Traffic benchmark [7] simulates a complex highway system (at a single car granularity), where the topology is a generic graph (specified via a configuration file), where nodes represent cities or junctions and edges represent the actual highways. Every node is described in terms of car inter-arrival time and car leaving probability, while edges are described in terms of their length. The highway's topology**,** in terms of its simulation objects**,** is distributed on a given number of LPs. Therefore, every LP handles the simulation of a node or a portion of a segment, the length of which depends on the total highway's length and the number of available LPs. Every LP simulates the cars which cross the junction or traverse the segment (depending on the

actual nature of the LP). Additionally, if the LP models a junction, the simulation model takes into account the entering paths, i.e. in a junction cars can enter the system according to some specific distribution. Cars can join the highway starting from cities/junctions only, and are later directed towards highway segments with some probability, depending on the traffic pattern which needs to be simulated.

Whenever a car is received by a LP, which will correspond to some *car-arrival* event scheduled for that LP, it is queued in the LP's list of traversing cars, and its speed (for the particular LP it is entering in) is determined according to a Gaussian probability distribution, the mean and the variance of which are specified at startup time. This allows capturing different speed limits which can be associated with different highway segments and the behavior of drivers in particular regions. Then, the model computes the time the car will need to traverse the node, adding traffic slowdowns which are again computed according to a Gaussian distribution. In particular, the probability of finding a traffic jam is a function of the number of cars which are currently passing through the node, and on the likelihood of accidents, which can be again specified according to some specific distribution**.** After having determined the exiting time from the node, a *car-arrival* event is scheduled towards the subsequent node to be passed through by the car, if any.

Traffic shows two peculiarities: On the one hand, there is a high coupling between the LPs, since the model intrinsically captures dynamics related to interactions across the LPs, which are naturally generated by the movement of vehicles within the highway; on the other hand, the events-exchange patterns across the LPs is somehow constrained, as compared to PCS, in fact there might be the case where from one LP an event (e.g., a *call-arrival*) can be sent only to a specific other LP, while PCS gives more statistical freedom. On the other hand with Traffic we

still have great opportunities for parallelism, since in a large highway, traffic evolution within a given zone is unlikely affected by the one in a different (far) zone. On the other hand, events such as accidents lead to concentrate the simulated evolution of the highway in a few LPs, which may lead to increase of the computing power needed for processing the simulation events for these LPs. As a consequence, the PDES environment should also in this case be able to react to such model execution dynamics in order to achieve effective exploitation of parallel execution schemes.

Current Architectural Trends: Multicores

The recent history in Computer Science has shown that, in order for software components to offer performance enhancements, the need for software optimizations were not necessarily a key factor. In fact, as *Moore's Law* [2] states, the total number of transistors on a microchip was doubling every 18-24 months. This electronic evolution was yielding a proportional increase in a single processor's clock speed, leading to an enhancement in computation efficiency which researchers, developers and users were receiving for free.

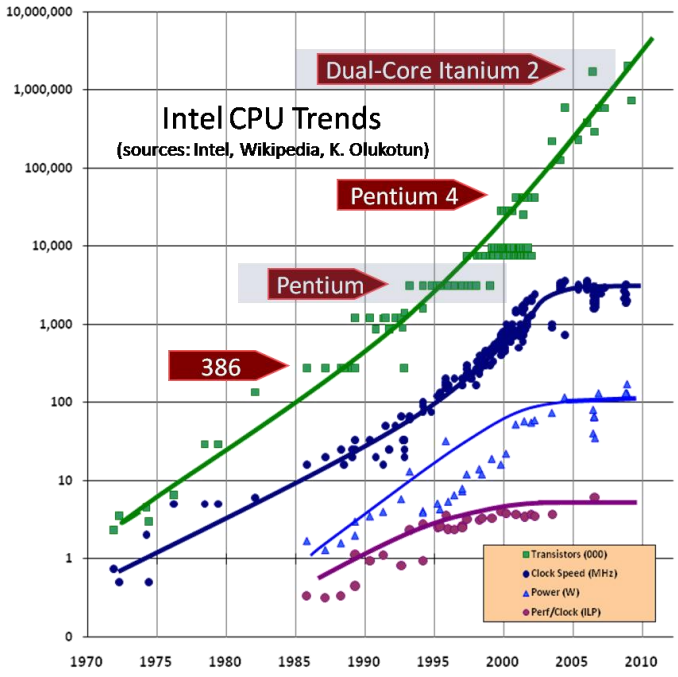

Figure 2 - Intel CPU Characteristics (updated August 2009)

(source: Herb Sutter, The Free Lunch is Over [3]*)*

Figure 2 shows (with reference to Intel processing units) how this trend has been almost the same until year 2003. Later on, although the number of transistors is still showing the same trend as before, clock speed increase has stalled. This is connected with the power curve: In fact, 130W of power consumption in a processing unit is considered an upper bound [3], which generates a *clock-frequency wall*. This is related to physical constraints, as the power consumption (P) is proportionally related to the frequency (f) [4]:

$$P = CV^2 f \qquad (3)$$

In fact, an increase in the clock frequency would produce unsustainable power consumption. Nevertheless, to face the continuously increasing demand in computing power, the industry's recent trend has brought the attention to multi/many-core architectures, where multiple processing units are packaged together into several interconnected processors. Of course, in order to efficiently rely on this emerging architecture, new programming models have been

developed. The High Performance Simulation field is just recently starting to look into these new possibilities, which are the actual topic of this chapter.

PDES in the Multicore Era

In literature, optimizations targeted at PDES systems mostly address the historical scenario, characterized by the aforementioned 1:1 mapping scheme, where a simulation-kernel instance is given a single processing unit for the execution of the simulation.

To exploit most of these optimizations, the traditional PDES paradigm could be easily mapped onto modern multi/many-core architectures while still relying on a 1:1 mapping between kernel instances and CPU-cores. However, this scenario would not allow the exploitation of all the potentials offered by the underlying architecture. For instance, although libraries targeted at supporting message passing are highly optimized on multicore machines, it would appear more promising to actuate cross-kernel communication by exploiting different supports like, e.g., shared memory and/or PDES-specific communication modules/schemes [5].

However, beyond the reshuffle of specific mechanisms to make them more suited for the underlying hardware's peculiarities, it looks convenient to relax the tight 1:1 mapping between kernel instances and processing units. In this way, a single kernel instance can run on top of multiple CPU-cores (just like it happens for the kernel of modern Operating Systems targeted at multi-core machines). In particular, a promising and general *symmetric* paradigm is to rely on a set of threads within the same simulation kernel instance (referred to as *worker threads*) which are not bound to a specific activity (e.g., message passing) but that can undertake any operation in the system (i.e., both event processing or housekeeping operations) depending on the actual activities that are required to carry on the overall simulation. This gives rise to maximal flexibility in terms of exploitation of available hardware resources. Therefore, to support this

paradigm, simulation kernels' internal architectures should be reshuffled by relying on multi-threading, which poses a set of issues to be carefully addressed. In what follows, we initially discuss architectural issues involved in this type of reorganization, as presented by some literature results [6]. Then, we focus on a specific and relevant aspect, which relates to how this architectural approach allows for an innovative way of (dynamically) getting a suited distribution of the simulation workload across the available CPU-cores.

Paradigm shift towards multi-threaded PDES

A paradigm shift towards the design/implementation of multi-threaded PDES kernels, each one entailing multiple, symmetric worker threads that can concurrently run any of the LPs hosted on top of the same kernel instance, is non-trivial. In particular, the following two key aspects must be carefully addressed:

1. avoiding synchronization phases while running housekeeping tasks in kernel mode to become a performance bottleneck;
2. avoiding loss of locality[1], which might (unacceptably) degrade the efficiency of the caching hierarchy.

As for point 1, there is one important difference depending on the execution mode of the simulation, namely *application mode* and *kernel mode*. In fact, when running in application mode (that is, when the actual simulation model is being run within a particular LP), worker

[1] *We explicitly refer here both temporal and spatial locality. In particular, if the same slice of data is repeatedly accessed along a given execution phase, then the overall performance will benefit from its residence in higher cache level. Accessing data along different threads may reduce the amount of relevant data kept in cache, and can therefore significantly affect the performance, due to increased memory-fetch time.*

threads inherently execute in *data separation*[2], considering that, as mentioned before in this chapter, LPs handle their own application-level data structures in the form of private/disjoint states. On the other hand, when running in *kernel mode* (that is, when the execution is under control of simulation-kernel modules for carrying on housekeeping operations*)*, great care must be taken to avoid performance degradation.

Specifically, in typical PDES platforms, the services and data structures used at kernel level to support the simulation's execution are very reduced in number if compared, e.g., to the ones used by common Operating System's kernels. It is therefore more likely that, in a concurrent kernel-mode execution of multiple worker threads contention and synchronization would easily become a bottleneck, if ad-hoc design mechanism were not employed. Such a problem might be exacerbated in medium/fine grain simulation applications, where control resides in application-level modules (for actual event processing) for limited WCT intervals, thus giving rise to frequent switches to kernel level housekeeping modules.

---

[2] *Data separation is a property of parallel/concurrent programs where, e.g., different threads can concurrently operate in isolation on a sub-portion of the overall data. This allows concurrent operations by a specific thread to be carried on regardless of the operations performed by other (concurrent) threads, without the presence of any critical section. This allows the parallel algorithm to produce correct results without relying on any locking primitive to protect critical sections, therefore giving rise to an improved execution speedup.*

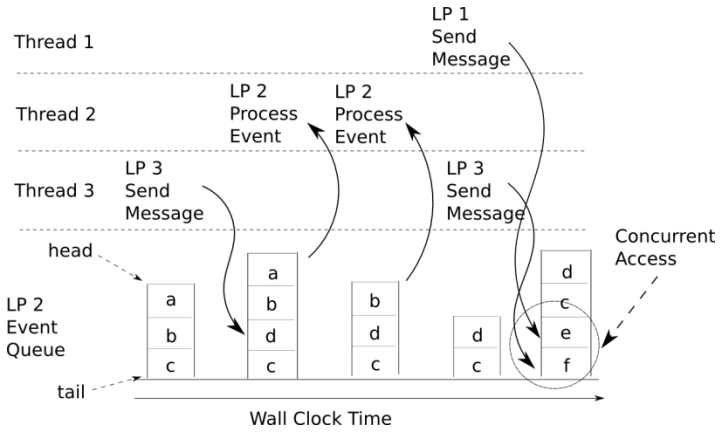

Figure 2 –Accesses to a Target Event Queue by Concurrent Worker Threads

Most notably, the data structures requiring frequent updates (to be performed coherently via proper kernel-level synchronization mechanisms) are the event queues of the LPs. Essentially, these data structures represent the core of cross-LP dependencies (we recall that the only way for different LPs to exchange information is message passing), thus involving update operations caused not only by activities executed by the worker thread currently taking care of running the *queue-owner LP*, but also by activities carried out by other worker threads. Figure 2 shows an example where two threads operating within the same kernel instance simultaneously attempt to deliver to the same event queue two new messages (two newly-scheduled events) for a given destination LP. These messages might have been produced along the execution path of the concurrent worker threads by two LPs still residing on the same kernel instance, which might have scheduled the corresponding events for the same destination as a result of local event processing activities.

Synchronizing accesses to the event-queues via a conventional locking mechanism would give rise to scalability problems. Also, it would give rise to critical sections whose duration would depend on the actual time-complexity of the queue update operation (which might even unpredictably depend on the specific event-timestamp pattern).

As for point 2, a symmetric multi-threaded simulation kernel allows virtual addresses related to

both application- and kernel-level data structures to be, in principle, accessible by any worker thread (since all the worker threads associated with the same simulation kernel instance operate within the same address space). However, such an unlimited access policy would cause frequent invalidation/refill of, e.g., the top-level private caches of individual CPU-cores, even when entailing processor affinity schemes involving the worker threads.

Addressing Kernel Level Synchronization

In order to reduce the synchronization costs while performing housekeeping operations, the architecture of the symmetric multi-threaded simulation kernel can be organized as recently proposed in [6].

According to the presented scheme, any housekeeping task potentially crossing the boundaries of individual LPs' data structures is dispatched according to the same rules employed to structure modern Operating System drivers, by organizing it according to *top/bottom-half* activities. More in detail, any of these tasks is considered as a logical interrupt to be eventually finalized within a bottom-half module. Hence, upon the interrupt occurrence, the task is not immediately finalized, and the target data structure is not immediately locked (or, the worker thread does not immediately enter a lock-waiting phase). Instead, a light top-half software module is executed, which registers the bottom-half function associated with the interrupt finalization within a per-LP bottom-half queue, resembling the Linux *task queue*.

The critical section accessing the bottom-half queue takes constant-time since each new bottom half associated with the LP is recorder at the tail of the queue. Also, when the bottom-half tasks currently registered for a given LP are finalized, the corresponding chain of records is initially unlinked from the corresponding bottom-half queue, which is again done in constant time by unlinking the head element within the chain from its base pointer.

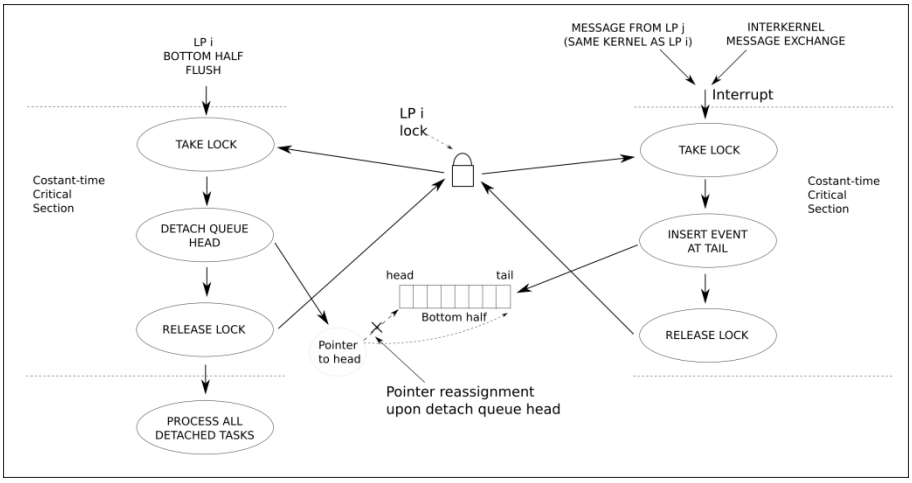

Figure 3 - Interrupt Handling Subsystem

A scheme related to the above architectural organization is provided in Figure 3. Operatively, the architecture can rely on a *spin-lock array*, having one entry for each LP hosted by the multi-threaded simulation kernel. The i-th entry is used to implement the critical section for accessing the bottom-half queue associated with the i-th LP hosted by the kernel, either for inserting a new bottom-half task, or for taking care of unlinking the current chain, in order to process and flush the pending bottom-halves. To reduce the performance impact generated by spin-lock accesses, techniques like the ones described in [6] can be used.

Overall, in this architectural organization, as soon as any worker thread becomes aware of a new message destined to the i-th locally hosted LP, it accesses the i-th bottom-half queue within a fast critical section that performs the insertion of the corresponding message delivery task. A similar situation occurs when the worker thread performs some receive operation via the messaging layer, which delivers a message incoming from some remote kernel instance and destined to a locally-hosted LP. As shown in Figure 3, the above circumstances are logically marked as interrupts, which will be finalized via the bottom-half mechanism.

Addressing Locality

In order to cope with locality, a promising approach investigated in [6] [7] is to devise the adoption of affinity mechanisms where any worker thread belonging to a given simulation-kernel instance is not allowed to run every LP hosted by that kernel at any time. Instead, it takes care of running a subset of these LPs, which are currently selected as being *affine* to the worker thread. In other words, for enhancing performance by locality-related means, temporary binding mechanisms should be adopted, which associate a subset of the locally hosted LPs to a specific worker thread. In this case, this worker thread is the only thread taking care of running these LPs during a specific WCT window.

Therefore, this worker thread should undertake the following activities to carry on the simulation:

- flushing the bottom-half queues associated with its affine LPs;
- dispatching these LPs for event execution in time interleaved mode.

The binding of a specific LP to a worker thread is not meant to be fixed, but can change over time, also in relation to variations of the amount of worker threads belonging to a given kernel instance, as it will be further discussed later on in this chapter.

Overall System Engineering Considerations

By the above description it is clear that a few additional engineering aspects need to be taken into account when designing/developing a symmetric multi-threaded PDES platform, when compared to a traditional single-thread organization. Here we recall the already pointed out, and provide additional hints on the actual engineering process. As said, the base for the support of the top/bottom half paradigm consists in having a non-blocking and asynchronous message notification system across threads within the same simulation kernel process. The channels

within the notification systems need to support multiplexing, with selective operations based on identifying messages with the ID of the destination LP. As illustrated, such a non-blocking service can be instantiated via fast (and constant time) critical sections protected by spinlock, which also avoid paying costs associated with systems calls access. On the other hand, other approaches could be employed such as Software Transactional Memory layers, which allow non-blocking atomic operations in a seamless manner.

As for exploitation of the caching system, (temporal) affinity between LPs and worker threads is only one of the relevant aspects. Another aspect relates to the avoidance of false cache sharing problems. Particularly, platform level data structures used to support housekeeping tasks should be explicitly separated per-thread and allocated in memory according to a cache aligned scheme. This can be achieved by either developing an ad-hoc memory allocator with the desired properties, or by relying on standard cache aligned allocators. Details on how to cope with these aspects can again be found in [6] [7].

Workload: Balance or Share?

The traditional approach to the design and implementation of optimistic simulation platforms consists in having – as mentioned before – multiple LPs being run within a single-threaded simulation-kernel process. Therefore, the typical literature approach aimed at achieving effective simulation runs, in terms of efficient exploitation of the available computing resources, is *load balancing (see, e.g.,* [8] [9] [10] [11] [12] [13]*)*, a schematization of which is depicted in Figure 4.

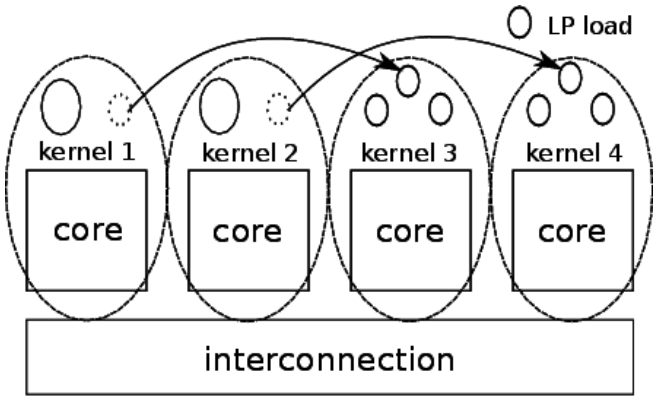

Figure 4 - Load Balancing

This technique is based on migration of the application load[4] amongst different simulation-kernel instances (i.e., user-space processes) while the run is in progress. It is clear that, given the tight 1:1 mapping between kernel instances and CPU-cores, no other means to dynamically rebalance the usage of resources can be employed, since each simulation-kernel process has a fixed amount of computing power allocated to it, namely one CPU-core.

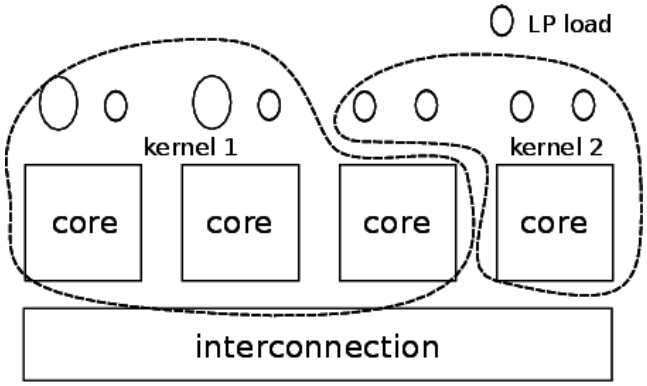

Figure 5 - Load Sharing

On the other hand, Figure 5 shows a different operational approach to maximize the fruitful resource utilization factor, which can be actuated thanks to the flexibility provided by the

[4] In the context of PDES, workload can be defined as the number and the granularity of pending events to be scheduled in the (near) future at each LP. Therefore, an *expected workload factor* can be associated with any LP according to this notion.

symmetric multi-threaded organization of the simulation kernel – namely *load sharing*. In this situation, the constraint mapping one simulation-kernel instance to one CPU-core is relaxed, allowing a single kernel instance to exploit a higher number of processing units to carry on the actual simulation run. This scenario enables a particular kernel instance to acquire/release computing resources, in the case of a workload factor increase/decrease by the hosted LPs. This can be achieved by scaling up/down the number of worker threads operating within each kernel instance. Recall that these worker threads operate symmetrically (not according to sector-specific functionalities), hence each of them is able to sustain the actual simulation workload (i.e., event processing).

This is clearly an orthogonal approach with respect to load balancing, considering that LPs are no longer migrated across various simulation-kernel instances. Additionally, the two approaches can be merged to achieve an even better simulation performance, when the execution of a simulation run involves several different machines in a cluster or a geographically-distributed system. In fact, while load sharing can be used to maximize the performance on a single node of the distributed environment, load balancing can be used at the same time to balance inter-node changes in the simulation workload.

This can be regarded as a promising approach, since it would allow making the best of the available computing power even in the case that the evolution of the system being simulated noticeably varies during the execution (giving rise to variations of the workload associated with individual LPs). This situation (unless for specific simulation model structures) can hardly be foresee before the actual simulation is put in place.

Advantages of Load Sharing vs. Load Balancing

As mentioned, load balancing relies on the migration of the workload from a given (overloaded)

kernel instance to a (more unencumbered) one. Migrating workload is not a trivial task, since this entails moving LPs' states across different simulation kernels, rerouting events in the whole platform, and presents some technical details which can become a non-negligible bottleneck in the execution of the overall simulation. Transparency issues also need to be considered since, unless for very limited cases where the simulation-kernel layer offers global memory-management schemes and automatic facilities for re-installing the exact memory layout of the migrating LP's state onto the destination kernel (see e.g. [14]), the intervention of the application-level programmer is requested. Particularly, he might be requested to explicitly provide the software modules for correct management of the memory layout and/or content of the LPs' states during migrations.

To provide more details on some of these aspects, the task of migrating the information required in order to support the execution of the migrating LP onto the destination kernel instance exhibits latency Δ_m, the lower bound of which can be expressed as:

$$\Omega(\Delta_m) = \delta_t \cdot \left[S_{state} + \sum_{i=1}^{N_P} S_{evt}^{i} \right] \quad (4)$$

where:

- δ_t is the average per-byte transfer time between source and destination simulation-kernel instances;
- S_{state} is the migrating LP's state size;
- N_P is the number of pending events for the migrating LP;
- S_{evt}^{i} is the size of the i-th pending event for the migrating LP.

With the above lower bound, we do not intend to capture aspects associated with, e.g., event-queue implementation and related scan/update costs. We do not even include the latency for

transferring data needed to support correct recovery in case of rollback, which is proper when considering an optimistically-synchronized PDES run. This data might entail, e.g., already-processed but not-yet-committed events – which might be required to be reprocessed in case of LP rollback after the migration phase – and state log information to correctly reconstruct past snapshots of the LP's state onto the destination kernel.

Anyway, by Equation (4), there is a clear dependency between the actual cost for supporting rebalance and the complexity of the simulation model, e.g., in terms of size of the state of individual LPs to be migrated. Also, in case of migration of multiple LPs, the above cost gets amplified (since it only expresses the per-LP migration overhead). Likely, this amplification could arise (or become relevant) in case of unbalanced execution scenarios of scaled-up models, possibly entailing a (very) large number of LPs.

On the other hand, for all the cases where load sharing is feasible, namely when aiming at the optimization of the (portion) of the simulation run hosted by a multi-core machine, the above migration costs can be eliminated at all. In fact, the only additional paid costs relate to worker-thread suspension/reactivation, which are anyhow not directly dependent on the aforementioned simulation model's complexity (e.g., in terms of state size of individual LPs).

A Load Sharing Model

To select the best-suited amount of CPU-cores to be destined for worker threads' execution in a given kernel instance during a specific WCT window, several load-sharing schemes can be envisaged. A highly-general and simple one, that has been introduced in [7], is presented in what follows. Let us suppose that we are in a setting where K simulation-kernel instances are present, running on C CPU cores (hosted by the same multicore machine). Let us denote with k_i, $i \in [1, K]$, an individual kernel instance, with $numLP^{k_i}$ the cardinality of the set of LPs hosted

by k_i, and with LVT_l^e the LVT associated with event e stored in the event queue of LP_l, $l \in [1, numLP^{k_i}]$.

Step 1. Each kernel instance k_i associates with each LP_l, $l \in [1, numLP^{k_i}]$, a *workload factor* L_l, defined as the WCT for advancing the LVT of LP_l by one unit. The workload factor L_l is computed by k_i on the basis of the total number of simulation events currently registered as to be processed within the corresponding event queue, and having a timestamp that falls within a given distance in the future, normalized to the LVT advancement they would produce, weighted by the average CPU time for event processing by LP_l, that is:

$$L_l = \frac{q_l \times \delta_l}{LVT_l^{q_l} - LVT_l^1} \qquad (5)$$

In Equation (5), q_l denotes the amount of pending events within the event queue of LP_l with timestamps that fall in the interval of interest, LVT_l^i is the timestamp associated with the i-th pending event along the event queue, and δ_l is the expected CPU requirement for event processing by LP_l along that chain of pending events. Among the above parameters, q_l and LVT_l^i are known in advance, since they are a function of LP_l's input queue's current state. Instead, δ_l is not known in advance, since it expresses the expected cost for events that have not yet been processed. Anyway, it can be approximated by using an exponential mean over already-processed events.

Step 2. k_i computes its total workload as:

$$L_k = \sum_{i=1}^{numLP^{k_i}} L_l \qquad (6)$$

Step 3. k_i determines the maximum degree of parallelism it can reach. This is done in relation to that each LP must execute events serially, i.e., no two worker threads can simultaneously take on

the execution of the same LP. Hence, the maximum degree of parallelism is determined by the number of knapsacks of LPs, such that the LPs within a same knapsack would globally induce the same workload as the LP associated with the highest workload factor. Obviously, one knapsack will consist of a unique LP, namely the one associated with the highest workload factor. This task is performed in several steps:

- Workload factors for the LPs hosted by k_i are non-increasingly ordered (let us call them in this order as $L_{l_1}, L_{l_2}, \dots, L_{l_H}$);
- As hinted, the first (i.e. the highest) factor L_{l_1} is taken as the reference value, and the knapsack formed by LP_{l_1} is defined;
- The other knapsacks are built by aggregating the remaining LPs according to a *0-1 one-dimensional multiple knapsack* problem solving algorithm. This is an NP-Complete problem, whose integral solution is non-trivial. However, a procedure can be followed where an ideally infinite set of J sacks, $J \in [1, \infty]$, is allowed to be used and for each of them, the greedy approximation approach proposed by George Dantzig [15] can be exploited, in which the constraint on the sack is released, by allowing a maximum "overflow" of 30%. At each step of the algorithm, $\forall i \in [2, H]$ the j-th knapsack's size K_{n_j} is updated as $K_{n_j} = K_{n_j} + L_{l_i}$, and thus considered full if the size constraint is violated. In that case, a "new" sack is created (i.e. j is increased) and begins to be filled, until the total workload is distributed across the sacks;
- The well suited number of worker threads required by simulation kernel k_i is $W^{k_i} = j$.

Step 4. k_i notifies the tuple $\langle W^{k_i}, L^{k_i} \rangle$ to a *master kernel*. Generally speaking, a master kernel can be either identified via a distributed consensus protocol, or can be known a-priori (e.g. by specifying it at compile time) depending on the actual simulation platform.

Step 5. The master kernel computes the total system's workload:

$$L^{tot} = \sum_{i=1}^{K} L^{k_i} \qquad (7)$$

Step 6. A preliminary estimation of the number of cores to be allocated to k_i is computed as:

$$T_{k_i} = \left\lfloor \frac{L_{k_i}}{L^{tot}} \cdot C \right\rfloor \qquad (8)$$

enforcing at least $T_{k_i} = 1$.

Step 7. A refined estimation is then calculated:

$$T'_{k_i} = \begin{cases} T_{k_i} & if\ T_{k_i} \leq W^{k_i} \\ W^{k_i} & if\ T_{k_i} > W^{k_i} \end{cases} \qquad (9)$$

Step 8. If $\sum_{i=1}^{K} T'_{k_i} < C$, then there are some CPU-cores still available. In this case, all the kernels are non-increasingly ordered by resource allocation reminder $R^{k_i} = \left(W^{k_i} - T'_{k_i}\right)$, and until there are CPU-cores still available, they are allocated in a round-robin fashion.

Step 9. The master kernel notifies to each kernel k_i the tuple $\langle j, T'_{k_j} \rangle\ \forall j$.

At the same time, the work in [7] also provides a model for determining the temporal binding of LPs to worker threads. Specifically, once the new amount of worker threads C_i to be employed by kernel k_i gets defined, a binding that allows balancing the whole workload related to local LPs onto the whole set of worker threads has been devised as follows. For the j-th LP hosted by kernel k_i, which we refer to as LP_i^j, the total amount of CPU-time cpu_i^j required for processing its events during the last observation period can be evaluated. Again, the maximum cpu_i^j value across all the locally hosted LPs represents a reference knapsack, and the corresponding LP_i^j is assigned to a given worker thread. Then, the approximated knapsack algorithm can be run, in order to determine which LPs must be assigned to the remaining worker threads.

A Case Study: Optimistic PDES

The optimistic PDES paradigm, as presented in [16], maintains the mapping of the simulation objects onto N uniquely-identified LPs, mapped in their turn onto K simulation-kernel instances. According to the optimistic synchronization protocol [16], events are executed regardless of their safety. This means that, whenever an LP is ready for processing (i.e. it has at least one pending event within its event queue), it can be dispatched, although some other LP in the system might eventually generate a new event which had to be executed before the currently dispatched one. This approach allows for great exploitation of parallelism while executing the simulation model, but might lead to *causal inconsistencies*, because events might be no longer executed in timestamp order.

The simulation kernel must therefore adopt some consistency-check procedure, in order to ensure that the final outcome of the simulation is correct. In particular, optimistic simulation platforms rely on a *rollback recovery scheme*, which can be based on periodically saving the state of every LP (incrementally or not). The taken state logs can be reloaded in case of causality inconsistencies, so to allow the resume of the LPs' evolution from a past, correct point along the simulation time axis.

Although this approach might seem cumbersome, it has been shown in literature to be incredibly effective, considering (among the other benefits) that it has been shown to provide run-time dynamics that are relatively independent of the simulation model's lookahead, and of the delay according to which new messages (i.e. new events) are reflected into the event queues of the destination LPs (namely, the message delivery delay).

Beyond discussing basic principles underlying the optimistic paradigm, the seminal paper in [16] also provides a reference architectural organization for optimistic simulation systems, which we

schematize in Figure 6. Specifically, we detail the suited set of data structures and functionalities/subsystems which should be provided in order to implement a platform relying on the optimistic paradigm[5].

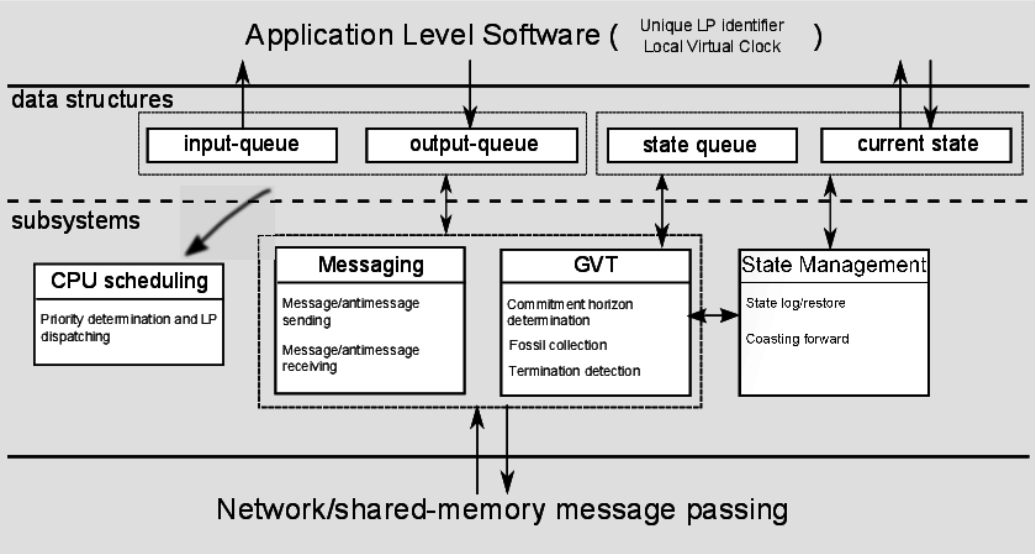

Figure 6 – Reference Architecture for Optimistic PDES Platforms

Input and output message queues are used to keep track of simulation events exchanged across LPs, or scheduled by an LP for itself. Typically, they are separated for different LPs, so to afford management costs. For the input queues, these costs are related to both event insertions and, e.g., event move from the past (already processed) part to the future (not yet processed) in case of rollback of a specific LP. The input queue is sorted by message (event) timestamps, while the output queue is sorted by virtual send time, which corresponds to the (future) LVT of the LP upon the corresponding event-schedule operation. As discussed by several works (see, e.g., [20]), the actual implementation of input queues can be differentiated (e.g. heaps vs. calendar queues), and possibly tailored to and/or optimized for specific application contexts, characterized by proper event-timestamp patterns (affecting the insertion cost depending on the algorithm used to

[5] By *subsystem* we just mean a logical differentiation, not an execution by a separate thread/process.

manage the queue). On the other hand, output queues are typically implemented as doubly-linked lists since insertions occur only at the tail (i.e. according to non-decreasing values of the LVT). Also, deletions from the output queues only occur either at the tail or at the head, the former occurring upon a rollback operation which undoes the latest computed portion of the simulation at each LP. In particular, all the output messages (i.e. the generated events) at the tail of the output-queue with send time greater than the logical time associated with the causality violation are marked, sent out towards the original destination in the form of *anti-messages* – used to annihilate previously sent messages and inform the original destinations of the occurred rollback[6] – and then removed from the output-queue. The latter are related to memory recovery procedures, which we shall detail later on in this section.

A *messaging subsystem* receives incoming messages from other simulation kernel instances, the content of which will be then reflected within the input queue of the destination LP. Also, it notifies output messages (i.e. newly scheduled events) to LPs hosted by other kernel instances, or the aforementioned anti-messages.

The *state queue* is the fundamental means for allowing a correct restore of the LP's state to a previous snapshot whenever a causality inconsistency is detected (i.e. the LP receives a message with timestamp lower than its current LVT, or an anti-message that annihilates an already processed event). The state queue is handled by the *state management subsystem*, the role of which is to save/restore state images, typically according to infrequent and/or incremental schemes (see, e.g., [17] [18] [19] [20]). Additional tasks by this subsystem are related to:

- performing rollback operations (i.e. determining what is the most recent suited state which has to be restored from the log);

[6] Chained rollback can arise if the received events have already been processed by the destination LPs.

- performing coasting forward operations (i.e. fictitious reprocessing of intermediate events in between the restored log and the point of the causality violation);
- performing *fossil-collection operations* (i.e. memory recovery) by getting rid of all the events and states logs which belong to an already committed portion of the simulation.

The *Global Virtual Time* (GVT) subsystem accesses the message queues and the messaging subsystem in order to periodically perform a global reduction aimed at computing the new value for the commit horizon of the simulation, namely the time barrier currently separating the set of committed events from the ones which can still be subject to a rollback. This barrier corresponds to the minimum timestamp of not yet processed or in-transit messages/antimessages. In addition, this subsystem cares about termination detection, by either checking whether the new GVT oversteps a given predetermined value, or by verifying some (global) predicate (evaluated over committed state snapshots [21]) which tells whether the conditions for terminating the model execution are met. Finally, this subsystem is also in charge of starting the aforementioned fossil collection procedure.

Additionally, a *CPU-scheduling approach* is used to determine which among the LPs hosted on a given kernel instance must take control for actual event processing activities. Among several proposals [17] [22], the common choice is represented by the Lowest-Timestamp-First (LTF) algorithm [23], which selects the LP whose next pending event has the minimum timestamp, compared to next pending events of the other LPs hosted by the same kernel. Variants for LTF exist, among which a basic (stateless) approach relies on traversing the next pending events across the input queues of all the LPs, and a recent stateful approach [24], which is based on reflecting variations of the priority of the LPs into the CPU-scheduler state, so that the LP with the highest priority can be determined via a query on the current CPU-scheduler state in constant

time.

Load Sharing in the context of Optimistic Simulation

In optimistic PDES runs, whenever a simulation kernel is hosting LPs with a load higher than the ones' being hosted by other instances, the LVT associated with LPs in the other instances can advance more. This is related to the fact that, when every kernel instance hosts the same number of LPs, a more overloaded instance must execute more events per WCT time unit. Since the advancement in LPs' LVT is just related to the number of events executed in a WCT time unit (we recall that events are impulsive), a less overloaded instance will process more events for all the hosted LPs, thus making their LVT advance more. This skew in the LVT values might induce, as a consequence, a higher amount of rollbacks (hence an increase in the wasted computation). In fact, the probability that LPs hosted by the overloaded simulation kernel would create causal inconsistencies in the less loaded LPs gets higher.

A load sharing system would be able to capture this imbalance, and assign a higher number of processing units to the more loaded simulation kernels. As a consequence, the LVT skew would be reduced, diminishing the amount of rollback operations and, in turn, increasing the overall efficiency of the simulation run.

Experimental evidence in relation to the above reasoning exists. Specifically, a set of measurements gauged from a real implementation of the innovative multi-threaded architecture and of the overviewed load sharing facilities within the open source ROme OpTimistic Simulator (ROOT-Sim) [25] [26] [27] are available.

The experimental data refer to tests carried out on a 64-bits NUMA machine, namely an HP ProLiant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 CPU cores (for a total of 32 CPU cores) that share a 10MB L3 cache

(5118KB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system that has been used in the experimental study is 64-bits Debian 6, with Linux kernel version 2.6.32.5.

For assessing the validity of the proposal, the previously presented two real-world application benchmarks have been used, namely *Personal Communication System* (PCS), a mobile phone simulator, and *Traffic*, a detailed highway simulator. Detailed descriptions for both these benchmarks are provided in [6] [7]. In the experimental study, the PCS benchmark has been configured in order to provide a constant workload across all the LPs during the whole simulation run. This has been done in order to measure the actual overhead of the load-sharing architecture, while not taking advantages from its ability to reallocate CPU-cores, given the constancy of the workload. In this assessment, 1024 wireless cells (where one cell is modeled by an individual LP) have been simulated, each one handling 1000 wireless channels.

As for the Traffic benchmark, in the experiments carried out, the whole Italian highway system has been simulated (with the exclusion of the islands) by relying on 1024 LPs.

In order to effectively assess the proposal, the experimental study has been based on the observation of two metrics, whose related data are shown in Table 1 and Table 2, namely *total simulation execution time* and *simulation speedup*. In the tables we report results for a serial (traditional) DES execution based on a calendar-queue scheduler (which constitutes the baseline for the evaluation of the speedup provided by the parallel runs), a single-threaded classical optimistic execution, and several multi-threaded executions with different amounts of kernel instances, giving rise to different multi-threading degrees within each kernel instance. Since the Traffic benchmark provides a more varying workload, in the corresponding table there is also a comparison with the orthogonal load balancing approach (still offered by ROOT-Sim according

to the description in [14]), while in the case of PCS this comparison is not shown, since that benchmark has been used solely for assessing the actual overhead of the presented architecture (with respect to a classical single-threaded execution), given that it exhibits balanced workload. The PCS benchmark's execution involved 830K events (overall), while the Traffic's one involved 400K events (overall).

Configuration	**Execution Time (seconds)**	**Speedup**
Serial	17000	–
Single-threaded	44	386
Multi-threaded (4k)	58	293
Multi-threaded (8k)	54	315
Multi-threaded (16k)	52	327
Multi-threaded (32k)	52	327

Table 1 - PCS experimental results

Configuration	**Execution Time (seconds)**	**Speedup**
Serial	17500	–
Single-threaded	54	324
Load Balancing	33	530
Multi-threaded (4k)	27	648
Multi-threaded (8k)	21	833
Multi-threaded (16k)	22	795
Multi-threaded (32k)	62	282

Table 2 - Traffic experimental results

By the results of the experimental assessment, it can be seen that for the PCS benchmark, the configuration with 4 kernels shows the highest overhead (expressed by a WCT increase for completing the run), which is in the order of 20%. Increasing the number of simulation kernels (each one entailing a reduced amount of worker threads) provides a 13%-15% overhead reduction. These results have been achieved for relatively fine event granularity (about 30 μs for

event processing), thus further supporting the viability of the proposal, since applications exhibiting coarser-grain events would absorb better the actual overhead induced by the symmetric multi-threaded architecture, e.g., in relation to the activities required for handling the top/bottom half mechanism, which is not present in the traditional single-threaded organization. Also, the parallel approaches provide super-scalar speedup with respect to the serial DES execution, which indicates that they are actually competitive.

As for the Traffic benchmark, the experimental study shows that the parallel approaches provide super-scalar speedup. All the multi-threaded versions of the simulation kernel provide speedup over the single-threaded one, which ranges in between 40% (for the 4 kernels configuration) and 55% (for the 8 and 16 kernels configuration). The execution with 4 kernel instances shows a reduced speedup due to several reasons:

- the rebalancing is more likely to map a worker thread on a core which is not actually sharing any level of cache;
- a worker thread can access remote memory with a higher probability (which, on NUMA machines, as the one used for the study, is very costly);
- worker threads are more subject to false cache sharing effects.

As for the execution with 32 multi-threaded kernels, the speed down is in the order of 15%. This is related to the fact that in this configuration no actual re-balancing is possible (in fact, each simulation kernel must have at least one worker thread in order to proceed in the simulation run). Therefore, this configuration again measures the architecture's overhead, with respect to the single-threaded organization, which is indeed comparable to the one shown when running the PCS (balanced) benchmark.

The last comparison is the one with respect to the load balancing configuration. Although we

note that this configuration provides speedup in the order of 40% with respect to the single-threaded approach, the delivered performance is comparable with the 4-kernels multi-threaded configuration, while the 8- and 16-kernel configurations of the multi-threaded architecture are still 30% faster than the load balancing configuration.

Conclusion

In this chapter we have discussed the issue of how to organize the architecture of PDES platforms in order to make them fully suitable for running on top of multi-core machines. This is a relevant aspect related to the possibility to improve the efficiency of these platforms (thanks to the fruitful exploitation of scaled up computing power), thus allowing them to increase the role they already have in engineering contexts where the timeliness of the delivery of simulation outputs plays a central role. We have presented general concepts related to the architectural organization, using the symmetric multi-threading paradigm as the base for instantiating an archetypal organization of the PDES platform. Then we have shown how the symmetric approach is naturally prone to the achievement of balanced exploitation of the computing resources offered by a multi-core machine. Finally, a real architecture of a PDES platform based on the above symmetric paradigm, and on the optimistic synchronization approach, has been presented as an example of real system leading to full, balanced exploitation of such computing power. This chapter is intended to provide contributions on the side of showing how the multi-core technological trend can be exploited for improving the timeliness according to which a simulation systems (in particular, a system based on the discrete event simulation paradigm) can support the execution general, complex and dynamic models.

Bibliography

[1] R. M. Fujimoto, «Parallel Discrete Event Simulation,» *Communications of the ACM,* vol. 33, n. 10, pp. 30-53, October 1990.

[2] G. E. Moore, «Cramming More Components onto Integrated Circuits,» *Electronics,* vol. 38, n. 8, pp. 114-117, April 1965.

[3] H. Sutter, «The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,» *Dr. Dobb's Journal,* vol. 30, n. 3, pp. 202-210, 2005.

[4] I. Corporation, *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor,* 2004.

[5] B. P. Swenson and G. F. Riley, "A New Approach to Zero-Copy Message Passing with Reversible Memory Allocation in Multi-core Architectures," in *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, 2012.

[6] R. Vitali, A. Pellegrini and F. Quaglia, "Towards Symmetric Multi-threaded Optimistic Simulation Kernels," in *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, 2012.

[7] R. Vitali, A. Pellegrini and F. Quaglia, "A Load-Sharing Architecture for High Performance Optimistic Simulations on Multi-core Machines," in *Proceedings of the 8th IEEE*

International Conference on High Performance Computing, 2012.

[8] A. Boukerche and S. K. Das, "Dynamic Load Balancing Strategies for Conservative Parallel Simulations," in *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 1997.

[9] G. D'Angelo and M. Bracuto, "Distributed Simulation of Large-Scale and Detailed Models.," *International Journal of Simulation and Process Modelling (IJSPM),* vol. 5, no. 2, pp. 120-131, 2009.

[10] D. W. Glazer and C. Tropper, "On Process Migration and Load Balancing in Time Warp," *IEEE Transactions Parallel Distrib. Syst.,* vol. 4, no. 3, pp. 318-327, 1993.

[11] C. D. Carothers and R. M. Fujimoto, "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms," *IEEE Transactions on Parallel and Distributed Systems,* vol. 11, no. 3, pp. 299-317, 2000.

[12] P. L. Reiher and D. R. Jefferson, "Virtual Time Based Dynamic Load Management In The Time Warp Operating System," *Transactions of the Society for Computer Simulation,* vol. 7, pp. 103-111, 1990.

[13] S. Meraji, W. Zhang and C. Tropper, "A Multi-State Q-Learning Approach for the Dynamic Load Balancing of Time Warp," in *Principles of Advanced and Distributed Simulation (PADS),* 2010.

[14] S. Peluso, D. Didona and F. Quaglia, "Application Transparent Migration of Simulation Objects with Generic Memory Layout," in *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*, 2011.

[15] G. B. Dantzig, «Discrete-variable extremum problems,» *Operational Research,* n. 5, 1957.

[16] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and System,* vol. 7, no. 3, pp. 404-425, July 1985.

[17] B. R. Preiss, W. M. Loucks and D. MacIntyre, "Effects of the Checkpoint Interval on Time and Space in Time Warp," *ACM Transactions on Modeling and Computer Simulation,* vol. 4, no. 3, pp. 223-253, July 1994.

[18] F. Quaglia, «A Cost Model for Selecting Checkpoint Positions in {Time} {Warp} Parallel Simulation,» *IEEE Transactions on Parallel and Distributed Systems,* vol. 12, n. 4, pp. 346-362, #feb# 2001.

[19] D. West and K. Panesar, "Automatic Incremental State Saving," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, 1996.

[20] R. Vitali, A. Pellegrini and F. Quaglia, "Autonomic Log/Restore for Advanced Optimistic Simulation Systems," in *Proceedings of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2010.

[21] F. Quaglia, «On the Construction of Committed Consistent Global States in Optimistic Simulation,» *International Journal of Simulation and Process Modelling,* vol. 8, n. 1, pp. 172-181, 2009.

[22] D. W. Bauer, G. Yaun, C. D. Carothers, M. Yuksel and S. Kalyanaraman, "Seven-O' Clock: A New Distributed GVT Algorithm Using Network Atomic Operations," in *Proceedings of the 19th Workshop on Parallel and Distributed Simulation*, 2005.

[23] T. Hamada and K. Nitadori, "190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[24] R. Ronngren, M. Liljenstam, R. Ayani and J. Montagnat, "Transparent incremental state saving in Time Warp parallel discrete event simulation," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, 1996.

[25] The High Performance and Dependable Computing Systems Research Group, "ROOT-Sim: The ROme OpTimistic Simulator - v 1.0," 2012. [Online]. Available: http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/.

[26] A. Pellegrini, R. Vitali and F. Quaglia, "The ROme OpTimistic Simulator: Core Internals and Programming Model," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, 2011.

[27] A. Pellegrini e F. Quaglia, «The ROme OpTimistic Simulator: A Tutorial,» in *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, Aachen, 2013.

[28] S. Kandukuri and S. Boyd, "Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications," *IEEE Transactions on Wireless Communications*, vol. 1, no. 1, pp. 46-55, 2002.