A Symmetric Multi-threaded Architecture for Load-sharing in Multi-core Optimistic Simulations

Roberto Vitali Alessandro Pellegrini Francesco Quaglia

DIIAG – Sapienza, University of Rome

Abstract

Parallel Discrete Event Simulation (PDES) is based on a simulation model partitioned into distinct Logical Processes (LPs) which are allowed to execute simulation events concurrently. We present here an innovative approach to load sharing on multi-core/multiprocessor machines for the optimistic PDES paradigm, where LPs can speculatively process simulation events with no a-priori verification of causal consistency, and violations (if any) are recovered via rollback techniques. Each simulation kernel instance, in charge of hosting and executing a specific set of LPs, runs a set of symmetric worker threads, which can be dynamically activated/deactivated on the basis of a distributed algorithm, which relies in turn on an analytical model providing indications on how to reassign processor/core usage across the kernels in order to efficiently handle the simulation workload. In order to optimize efficiency and reduce lock-release phases used to synchronize the threads when running in kernel mode, we propose to borrow from operating systems theory and readapt the top/bottom-halves paradigm to the design of load-sharing oriented optimistic simulation systems. We also present a real implementation of the our load sharing architecture within the ROme OpTimistic Simulator (ROOT-Sim), namely an open-source C-based simulation platform implemented according to the PDES paradigm and the optimistic synchronization approach. Experimental results for an assessment of our proposal are presented as well.

1 Introduction

Parallel Discrete Event Simulation (PDES) techniques are well known for being a classical means to develop simulation systems featuring high performance, which is essential for differentiated contexts, such as symbiotic systems or simulation-based (time-critical) decision making. The basic idea underlying these techniques is to partition the simulation model into several distinct objects, also known as Logical Processes (LPs) [Fuj90], which concurrently execute simulation events on clusters, SMP/multi-core machines and/or even desktop grids [PF08].

The main problem in the design/development of this type of simulation platforms is synchronization, the goal of which is to ensure causally-consistent (e.g. timestamp-ordered) execution of simulation events at each concurrent LP. In literature, several synchronization protocols have been proposed, among which the optimism-oriented ones (e.g., the Time Warp protocol [Jef85]) are highly promising. With these protocols, block-until-safe policies for event processing at the LPs are avoided, thus allowing speculative computation, and causal consistency is guaranteed through rollback/recovery techniques, which restore the system in a correct state upon the a-posteriori detection of consistency violations. These are originated when LP_a schedules a new event destined to LP_b having a timestamp lower than the one of some event already processed by LP_b. This approach has been shown to exhibit performance which is relatively independent of both (a) the lookahead of the simulation model and (b) the communication latency between the concurrently running LPs (e.g. concurrently running simulation kernel instances). It is therefore viable and effective for a wide spectrum of both application-specific and infrastructure-related settings.

Multiple LPs typically run within a same simulation-kernel process, and according to the classical organization (see, e.g., [CBP00]), this process is single-threaded. As a consequence, the hosted LPs are dispatched and run on top of an individual CPU-core according to a classical time-interleaved mode resembling what happens in traditional Operating Systems targeted at single-core machines. By this organization, the typical literature approach aimed at achieving effective parallel/distributed simulation runs, by optimizing the exploitation of the available computational resources, is *load balancing*. This technique is based on migrating the application load (i.e., LPs) amongst different simulation-kernel instances (i.e. different processes). In other words, the only means to dynamically re-balance the load is to explicitly re-map the LPs across the kernels, since each kernel instance has a fixed computational power (i.e. one CPU-core) allocated to it.

We hereby propose an orthogonal technique targeted at optimistic PDES system run on top of multi-core machines, which is based on computational power (expressed in terms of CPU-cores) dynamical reallocation over time towards the different active simulation-kernel instances. Hence, we allow a dynamical scale up/down in the number of symmetric worker threads belonging to each kernel instance, which sustain the execution of the whole set of LPs locally hosted by that same instance, depending on whether LPs (dynamically) increase/decrease their computational power demand. Overall, the load sharing approach is pursued, in terms of dynamical redistribution of the whole simulation load across the set of available computing resources. Further, dynamical scale up/down of the amount of worker threads per simulation kernel instance is based on an innovative analytic approach that we also propose in this paper.

From the architectural perspective, what we provide is a paradigm shift towards the building of simulation-kernels for optimistic PDES systems entailing scheduling and execution functionalities typical of modern Operating Systems' technology, in particular the one targeting SMP and/or multi-core machines. However, such a shift is non-trivial, since (optimistic) simulation platforms are typically expected to expose a reduced set of services (compared, e.g., to those offered by a conventional Operating System kernel), internally handled by the simulation-kernel layer via a relatively reduced set of data structures. Hence, the likelihood of conflicts in the access to these data structures upon simultaneous execution in kernel mode by multiple worker threads (each running on top of a different core) may easily become a bottleneck. To address this issue we borrow from the top/bottom-half programming paradigm, currently employed for handling interrupts within modern, multi-core Operating Systems, to design multi-threaded optimistic simulation kernels guaranteing minimal length of wait-for-lock phases, and high scalability.

Beyond general design indications, we also present a real implementation of our architectural proposal within ROOT-Sim, namely an open-source optimistic simulation platform [QPV11b] implemented according to the optimistic synchronization paradigm. Experimental data assessing the viability and the relevance of the provided approach are also presented.

2 Related Work

Given that we aim at optimizing the use of the available computing resources in face of dynamism and fluctuations of the actual workload associated with the LPs involved in the run, our work is naturally related to all the literature solutions that have presented policies for load balancing in the context of either conservative (e.g. [BD97]) or optimistic simulation (e.g. [GT93]). As already hinted, the main differences between our proposal and these works are in that (A) we reassign the computational power, instead of the workload, across the active simulation kernel instances, and (B) we rely on an innovative, multi-threading oriented paradigm in order to exploit dynamically scaled up/down power available to each kernel instance. Also, our proposal can be considered as *orthogonal and complementary* to the above results, when considering that we target multi/many-core machines, while the aforementioned load-balancing schemes can be used for load-redistribution on distributed memory systems (e.g. clusters).

As for HLA-based simulation platforms (see, e.g., [MW95]), multi-threading has been used to implement non-blocking interoperability services across federations of simulators. The strongest difference from our approach is that multi-threading has been used to implement sector-specific functionalities, while we use it as a means to overtake differentiated operations (including event processing), depending on dynamical variations of the application level workload. In addition, to the best of our knowledge, changes in the number of worker threads has never been used to perform dynamical optimizations in response to workload's variations. It has only been employed in master-slave simulation architectures to cope with dynamical increase/decrease of the amount of available computing resources (e.g. for simulation platforms running on top of desktop grids [PF08]). However, in such a context, concurrent threads operate on inherently partitioned data, while we approach multi-threading in the presence of shared (e.g. kernel-level) data structures.

When considering solutions specifically oriented to improve the performance of simulation platforms on multi-core machines, one approach having relations with our proposal can be found in [LW11]. However, this approach is targeted at a specific architecture, namely the IBM cell processor, while our proposal is general, thus being suited for differentiated multi-core platforms. Also, the work in [LW11] is oriented to optimize the simulation via task parallelization schemes that are orthogonal to the power reallocation scheme we present in this article. Similar considerations can be made for other works which address the issue of improving the performance of simulation systems via the exploitation of hardware parallelism offered by GPU architectures (see, e.g., [HN10]). These approaches are mostly suited for data parallelism while we deal with more general parallelism schemes, which are proper of the PDES paradigm. Also, dynamic power reallocation across different simulation kernel instances is not targeted by those works.

The work in [lCsLpY^{+}11] has recently presented an approach for improving optimistic simulations on multi-core machines via the employment of a global schedule mechanisms relying on a distributed event queue. This proposal has been evaluated with a multi-

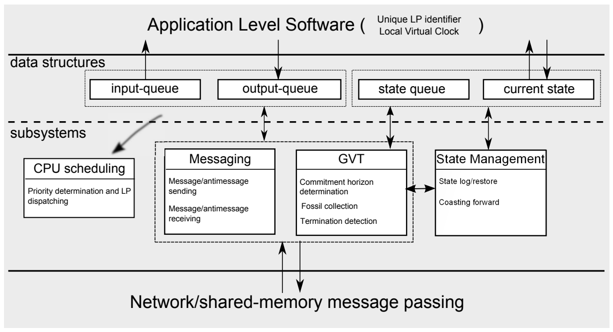


Figure 1: Reference Architecture for Optimistic Simulation Systems.

core machine comprising 8 cores. Differently from this work, our proposal targets the traditional case of local schedule, characterized by higher scalability thanks to the avoidance of cross-kernel synchronization operations while handling scheduling tasks, as we also show via experimentation on a multi-core server equipped with a total of 32 cores. Similar considerations can be made when considering simulation architectures like ThreadedWarped [Mil10], which uses a global priority queue.

3 Design Indications

3.1 Optimistic PDES Systems' Fundamentals

In the optimistic PDES paradigm, LPs are allowed to process their simulation events without preliminary verifying that these events do no violate causality rules. Rather, as soon as new events get available for a specific LP, they are eligible for processing. This allows for great exploitation of parallelism, since no block-until-safe rules are adopted. On the other hand, in case a causality violation is revealed, due to the arrival of an event for a destination LP that has already processed other events with larger timestamps (namely events in the future of the arriving one), this LP is rolled back to a correct state value. Also, the effects on other LPs associated with events that were produced by the rolling back LP during the causally incorrect portion of the simulation are also undone by the simulation-kernel via proper messaging operations. In Figure 1 we show the reference architectural organization for optimistic simulation systems, as depicted by the seminal paper in [Jef85].

Input and *output message queues* are used to keep track of simulation events exchanged across LPs, or scheduled by an LP for itself. These are typically separated for different LPs, so to afford management costs. For the input-queues, these costs are related to both even 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 local virtual clock of the LP upon the corresponding event-schedule operation. As discussed by several works (see, e.g., [RA97]), 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 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 local virtual clock). Also, deletions from the output-queues occur either at the tail or at the head, but not at arbitrary positions. Tail deletions occur upon a rollback operation involving the corresponding LP, which undoes the latest computed portion of the simulation at the LP. In particular, all the output messages (i.e. the scheduled 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, and then removed from the output-queue. Anti-messages are used for annihilating previously sent messages so to notify to the destinations about the rollback occurrence at the source LP. Upon their arrival, chained rollback can be generated in case the events carried by the originally sent messages have already been processed by the destination LPs. Head deletions from the output-queues are related to memory recovery procedures, which we shall detail later on in this section.

A *messaging subsystem* takes care of receiving incoming messages from other simulation kernel instances, whose content will be then reflected within the input queue of the destination LP. Also, it is used to notify 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 state to a previous snapshot whenever a causally inconsistency is detected (i.e. the LP receives a message with timestamp lower than its current simulation clock, or an anti-message that annihilates an already processed event) ([1]). The state queue is handled by the *state management subsystem*, the role of which is to save/restore state images. Additional tasks by this subsystem are related to (i) performing rollback operations (i.e., determining what is the most recent suited state which has to be restored from the log), (ii) performing coasting forward operations (i.e., fictitious reprocessing of intermediate events in between the restored log and the point of the causality violation) and (iii) 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). Several solutions have been presented in literature for optimizing the performance of the state-save/restore subsystem (see, e.g., [PLM94, Qua01, QS03, RA94]) and/or for providing state-log/restore transparency vs the application layer (see, e.g., [RLAM96, TQ08]).

[1]Recently, approaches have been provided which substitute or complement log based state recoverability via reverse computation techniques [CPF99]. In such a case, recoverability also relies on having a reverse version of the application code, which is able to backward apply changes occurred on the LP state since the point of the causality violation.

The latter aspect is crucial for the usability of the optimistic paradigm, since it relates to relieving the application programmer from the burden of designing/implementing low level state management tasks.

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 events. 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) which tells that the conditions for termination model execution are met. Finally, this subsystem is also in charge of performing the so-called fossil collection procedure, aimed at recovering memory buffers currently keeping obsolete messages end logs, namely those related to the newly committed portion of the computation. As for GVT computation, termination detection and fossil collection, the literature also offers a plethora of optimized solutions, which are either general purpose or tailored to specific computing platforms such as shared memory systems vs clusters (see, e.g., [BYC+05, FH97]).

Finally, a central point relates to the CPU-scheduling approach used to determine which among the LPs hosted by a given simulation-kernel instance must take control for actual event processing activities. Although several proposals have been made, the common choice is represented by the Lowest-Timestamp-First (LFT) algorithm [LL91]. It selects the LP whose pending next-event has the minimum timestamp, compared to pending next-events of the other LPs hosted by the same kernel. Coupled with the traditional single-threaded approach for the implementation of the simulation kernel, LTF has the advantage of avoiding the generation of causality violations across the LPs hosted by the same kernel instance. This is because these LPs are dispatched in a similar way to what would happen on top of a sequential simulation engine, which imposes a timestamp-ordered sequence of CPU-schedule operations for all the events (across all the LPs). Hence, rollbacks can be generated only in relation to events scheduled between LPs hosted by different kernels, which contributes to reduce the amount of rollback. Different design/implementation variants for LTF exist (see, e.g., [SQ10] providing different time complexities).

4 The Symmetric Multi-threaded Architecture

4.1 Handling Kernel-level Synchronization

A paradigm shift towards the design/implementation of symmetric multi-threaded optimistic simulation kernels, entailing multiple worker threads that can concurrently run any of the LPs hosted on top of the same kernel instance, needs to avoid synchronization phases while running in kernel mode to become a performance bottleneck. Specifically, while different worker threads inherently execute according to data partitioning paradigms once entered application mode (since, in accordance with the specification of the original Time Warp protocol [Jef85], each logical process handles its own application-level data structures), care must be taken to avoid "*lock-everything effects*" when running in kernel mode. The risk for these effects is actually due to the reduced set of subsystems forming the optimistic simulation kernel (compared, e.g., to those typically included within the kernel of a general purpose Operating System), and also to the inherent strict coupling among the LPs (compared, e.g., to the typical level of coupling of different processes running on top of a conventional Operating System).

Most notably, the data structures requiring frequent updates, to be performed coherently via proper kernel-level synchronization mechanisms, are both the input and output queues of the LPs. Essentially, these data structures represent the core of cross-LP dependencies, thus involving update operations caused not only by the activities executed by the worker thread currently taking care of running the "queue-owner LP", but also by the activities carried out by worker threads taking care of running other LPs. Synchronizing the access to these data structures via a conventional locking mechanism would give rise to scalability problems, exactly due to such a strict coupling. Further, it would give rise to critical sections whose duration would depend on the actual time-complexity of the queue-update operation.

We note that the access to the LPs' state queues (either for saving or restoring a state image) does not induce thread synchronization issues since the need for state log/recovery operations is only an indirect reflection of cross-LP coupling, caused by events scheduled across the LPs. In other words, a single worker thread is allowed to safely operate on the LP's state queue at any time, namely the worker thread that has taken care of dispatching that LP for either forward or rollback execution.

The architectural organization we propose in this paper to cope with the reduction of synchronization costs while performing housekeeping operations borrows from the design principles proper of multi-processor/multi-core Operating Systems. Specifically, 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. Hence, whenever the need for the execution of such a task arises, it (logically) takes place as an interrupt to be eventually finalized within a bottom-half module. More in details, upon the interrupt occurrence, we do not immediately finalize the task, thus not immediately locking (or waiting for the lock) on the target data structure. Instead we simply execute a light top-half

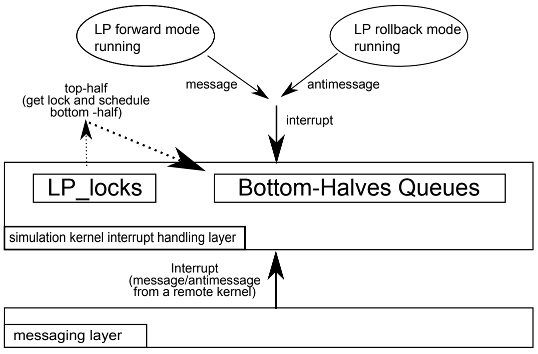

LP forward mode running
LP rollback mode running
top-half (get lock and schedule bottom -half)
message
antimessage
interrupt
LP_locks
Bottom-Halves Queues
simulation kernel interrupt handling layer
Interrupt (message/antimessage from a remote kernel)
messaging layer

Figure 2: Top/Bottom-Halves Architecture within the Symmetric Multi-threaded Optimistic Kernel.

module which registers the bottom-half function (and its parameters) 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 flushed, 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 ([2]). Given that the access to the LP bottom-half queue represents in our architectural organization the only frequently occurring synchronization point, constant-time for the corresponding critical sections directly leads to minimizing kernel level synchronization costs.

The schematization of our proposal is presented in Figure 2. Basically, our approach can be supported by relying on a spin-lock array, named `LP_LOCKS`, having one entry for each LP hosted by the multi-threaded simulation-kernel. `LP_LOCKS[i]` is used to implement the critical section for the access to the bottom-half queue associated with the i-th LP hosted by the kernel, either for inserting a new bottom-half task to be eventually flushed, or for taking care of unlinking the current chain, in order to flush the pending bottom-halves.

Let us now depict when (logical) interrupts to be handled via this type of organization occur. Basically, an interrupt occurs as soon as any worker thread currently active within the symmetric multi-threaded kernel becomes aware of a new message/antimessage destined to the i-th locally hosted LP. In such a case, the worker thread needs to accesses the i-th bottom-half queue within a critical section that performs the insertion of the corresponding message/antimessage delivery task, as explained above. To provide additional details, awareness by a worker thread of a new message/antimessage destined to a locally hosted LP arises in three different circumstances:

[2]Actual data structure updates are not performed within the critical section, but are anyhow safe since, as it will be discussed in Section 4.2, for locality reasons we will allow a single worker-thread at any time to be in charge of flushing the bottom-halves of a given LP.

(i) The worker thread is currently running the locally hosted LP_j in forward mode, and this LP produces a new event to be scheduled for the locally hosted LP_i. Thus the worker thread enters kernel mode for actuating the delivery of the corresponding message to LP_i's input-queue. (Note that j might be equal to i, hence giving rise to the case where sender and receiver coincide.)

(ii) The worker thread is currently running the locally hosted LP_j in rollback mode (hence it is performing kernel level housekeeping operations associated with revealed causality errors), which gives rise to the production of an antimessage destined to LP_i, which requires access to LP_i's input queue for annihilating the original message. (Again we might have $j = i$.)

(iii) The message passing layer notifies the worker thread (e.g. via an explicit message receive operation executed by this thread according to a traditional polling scheme) about a new message/antimessage incoming from some remote kernel instance.

As shown in Figure 2, we logically mark all the above three circumstances as interrupts, which will be treated homogeneously, and whose associated message/antimessage delivery operation will be finalized via the bottom-half mechanism.

We note that spin-locks may anyhow exhibit non-minimal costs since they require the corresponding operations to be performed via sequences of atomic instructions (e.g. via the `LOCK` prefix for the IA-32 instruction set). Additionally, since they are shared and accessed by different threads, cross-cache invalidation effects can be induced as soon as one worker thread gains control on the spin-lock. To reduce these effects, we have devised the presence of an additional array of flags `LP_FLAGS` (see again Figure 2), where `LP_FLAGS[i]` indicates whether the corresponding bottom-half queue, namely the one associated with the i-th locally hosted LP, is not empty. Actually, `LP_FLAGS[i]` gets updated within a critical section protected by `LP_LOCKS[i]`, either when a new bottom-half is inserted within the corresponding queue (in this case the flag is raised), or when the queue is flushed (in this case the flag is reset). However, `LP_FLAGS[i]` is also accessed before trying to lock the bottom-half queue in order to avoid spin-lock operations in all the cases where the queue would reveal empty once accessed within the critical section leading to flush operations. The exact scheme looks therefore as follows:

```
TOP-HALF:                      BOTTOM-HALF:
 lock(&LP_LOCKS[i]);            if (LP_FLAGS[i]){
 <log bottom-half>;               if (try_lock(&LP_LOCKS[i])){
 LP_FLAGS[i] = TRUE;                 <unlink bottom-halves>;
 unlock(&LP_LOCKS[i]);               LP_FLAGS[i] = FALSE;
                                     unlock(&LP_LOCKS[i]);
                                     <perform bottom-halves>;
                                  }
                                }
```

Being `LP_FLAGS[i]` checked non-atomically wrt lock acquisition when attempting to perform bottom-halves, we might experience false negatives in case the top-half finalizes the insertion of the bottom-half task concurrently with the check. However, this does not represent a safety problem since the flag will be rechecked periodically in subsequent attempts to flush the corresponding bottom-half queue, thus eventually falling within the case where the bottom-half queue is correctly reflected into the state of the input queue of the destination LP. Such a reflection might therefore experience only a delay, which resembles delays introduced by traditional single-threaded kernels while reflecting the content of cross-kernel messages into the system state, which is typically affected by the polling period according to which the messaging layer is accessed for acquiring not yet delivered messages. Further, as hinted in footnote 1, a single worker thread at a time will be allowed to manage flush operations for a given LP, hence no false positives will ever be experienced.

As a last note, messages/antimessages whose deliveries are still pending, being them recorded as tasks to be finalized within bottom-half queues, represent a kind of in-transit data, whose timestamp needs to be accounted for when computing the GVT value.

4.2 Tackling Locality Issues

Given that all the worker threads associated with the same simulation kernel instance operate within the same address space, the symmetric multi-threaded kernel allows virtual addresses related to both application and kernel level data structures (associated with whichever LP) to be, in principle, accessible by any worker thread. However, such a level of sharing would cause frequent invalidation/refill of, e.g., the top-level private caches of individual cores, even when entailing processor affinity schemes involving the worker threads. As an example, data structures associated with an LP that has been lastly accessed by a given worker thread would be flushed by the corresponding private caching system upon the first write access by a different worker thread.

Overall, while developing a symmetric multi-threaded optimistic simulation-kernel a core additional issue to address is related to maintaining an adequate level of locality, so to avoid harming caching performance. In order to cope with this issue, we devise the adoption of affinity mechanisms such that a worker thread belonging to a given simulation-kernel instance is not allowed to run every LP hosted by that kernel. 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, we devise the use of temporary binding mechanisms associating a subset of the locally hosted LPs to a specific worker thread, which is therefore the only thread taking care of running these LPs during a specific wall-clock-time window. We note that this approach resembles what is done by the scheduler of Linux kernel 2.6, where a temporary binding of active processes/threads to a specific CPU-core is supported for both (a) locality and (b) reduction of the CPU scheduling cost.

Overall, within the affinity scheme, each worker thread is in charge of:

(i) Flushing the bottom-half queues associated with its affine LPs, which is executed periodically according to a traditional polling approach.

(ii) Dispatching its affine LPs for execution in time interleaved mode.

We note anyway that 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 activated within a given symmetric optimistic kernel instance. The policy according to which the locally hosted LPs are reassigned to the worker threads will be discussed in Section 5, together with the performance model we use to reallocate the computational power (and hence CPU-cores) to the different symmetric simulation-kernel instances.

As an additional note in relation to locality, we also devise proper memory layout mechanisms in order to reduce the false cache sharing problem for kernel-level data structures. As an example, the entries of both the `LP_LOCKS` and `LP_FLAGS` arrays, which represent frequently accessed synchronization data structures, can be memory bind to different cache lines, which can be easily achieved by exploiting, e.g., the `posix_memalign` API plus padding schemes. The same approach can be taken for the meta-data associated with each single LP hosted by each instance of the symmetric multi-threaded kernel, so that once an LP is bind to a given worker thread, cache interference due to accesses to meta-data does not arise.

5 Computational Power Reallocation Policies

The symmetric multi-threaded kernel allows scaling up/down the amount of per-kernel worker threads without any change in the internal operating mode. This allows for dynamically reallocating the computational power (in terms of CPU-cores) to the active kernel instances depending on fluctuations of the workload and efficiency variations within the optimistic simulation run. In this section we first provide an approach for reallocating the CPU-cores to the active kernels. Then we address the issue of (temporarily) binding the LPs hosted by a given simulation kernel instance to specific worker threads.

5.1 Dynamical Assignment of CPU-Cores to Kernels

Let us denote with C_{tot} the amount of available CPU-cores, and with $K_{tot} < C_{tot}$ the number of active symmetric multi-threaded kernel instances (the case $K_{tot} = C_{tot}$ trivially boils down to the traditional scenario where each kernel instance is allowed to run on a single CPU-core, hence in the typical single-threaded mode). Our first objective is to determine the amount of CPU-cores C_i (with $1 \leq i \leq K_{tot}$) to be assigned to kernel instance K_i for a given wall-clock-time window, so to improve resource exploitation for fruitful processing activities.

In our proposal, the re-evaluation of C_i values can be carried out periodically, for example upon computing a new GVT value or after a set of subsequent GVT computations. This also allows to exploit a set of metrics characterizing the parallel simulation run, as an example in terms of determination of the event rate (committed events per wall-clock-time unit) achieved by each of the symmetric multi-threaded kernel instances. We denote the event rate achieved by kernel K_i as evr_i. This quantity is a measure for the fruitful (non-rolled back) amount of simulation work carried out by each kernel instance. In an ideal scenario where the efficiency is maximized (i.e. where the undone computation is negligible), each symmetric multi-threaded kernel instance K_i should use an amount of computational power that suffices to execute exactly evr_i events per wall-clock-time unit. In fact, an excess of computational power could lead to over-optimism and hence to rolled back computations, thus moving the run-time dynamics far from the above depicted ideal case. So the idea behind the determination of C_i values is to dynamically assign an amount of CPU-cores to kernel K_i which is proportional to the actual computation requirements of K_i for the achievement of its relative event rate, compared to the one by the other kernels. Actually, to also take care of the real CPU requirements on a given kernel instance (so to also take into account possible variance of the event granularity across the LPs hosted by different kernel instances), which is the indicator of the real usage of computational power for committing the events, the evr_i metric can be refined by weighting it via the average CPU time required for processing the events on a specific kernel K_i, which we denote as Δ_i. Hence we express the weighted event rate as $wevr_i = evr_i \times \Delta_i$.

In other words, $wevr_i$ values observed during the last wall-clock-time period express the relative CPU requirements of each kernel instance in order to carry out productive simulation work, in relation to the activities of the other kernels and the outcoming synchronization dynamics. Hence, assigning a computational power proportional to the relative weighted event rate would tend to lead to the situation where each kernel instance advances its LPs in simulation time in a “synchronization suited” manner according to what the other kernels are able to do on their own. This part of the dynamic reallocation scheme would therefore tend to avoid significant presence of overoptimistic kernel instances during the various phases of the run.

It is anyway typical that performance can be further enhanced even in cases where the efficiency is already maximized (or optimized), for example by further reassigning the computational power depending on the real weight of the workload associated with the hosted LPs. As an example, for loosely synchronized models we may have two or more groups of LPs that do not interact, or stop interacting during the run (hence eventually not directly impacting synchronization and efficiency), exhibiting different speed of advancement in simulation time due to, e.g., different weights of the corresponding events in terms of CPU requirements. In such a case, the completion of the simulation would be delayed by the slowest group. Therefore, within the dynamic scheme for resource assignment, an increase of computational power should also be envisaged for all those kernel instances exhibiting larger CPU requirements to advance in simulation time. To this end we include in our scheme the parameter $wcta_i$, which indicates the wall-clock-time required by kernel K_i to advance a single simulation time unit. The usage of this parameter within the dynamic reallocation scheme would tend to complement the above described one by further attempting to align the advancement of the different symmetric multi-threaded kernel instances in simulation time while the run proceeds.

Finally, the amount of cores C_i to be assigned to kernel K_i should anyway be bounded by the maximum degree of parallelism that can be accomplished by K_i, which is a function of the amount of locally hosted LPs. In fact, each LP is an intrinsically sequential entity, which is not further parallelized, thus not being allowed to simultaneously use multiple CPU-cores for its execution.

Overall, we devise the following rules for dynamically defining the amount of CPU-cores to be reassigned to each kernel K_i in order to optimize the usage of the available computational power:

1. For each kernel K_i the parameter $\alpha_i = \frac{wevr_i}{\sum_{j=1}^{K_{tot}} wevr_j}$ is computed.

2. A first calculation of C_i is then performed as $C_i = \lfloor \alpha_i \times C_{tot} \rfloor$.

3. For each kernel instance K_i for which the condition $C_i \geq numLP_i$ is verified (where $numLP_k$ identifies the number of LPs hosted by K_i), then C_i is definitively set to $numLP_i$. In fact, additional CPU-cores could not be effectively exploited for parallelization of the locally hosted LPs.

4. At this point, there could be some CPU-cores left to be assigned, which we decide to assign on the basis of (A) the request for allocation remainder of kernel K_i, namely $r_i = [(\alpha_i \times C_{tot}) - C_i]$ and (B) the parameter $wcta_i$. In particular, we order the kernels for which the finalization of C_i values still needs to be performed (so the ones already finalized in point 3 are excluded) according to decreasing values of the product $r_i \times wcta_i$, and we assign the remaining CPU-cores according to a round-robin rule following the priority defined by such an ordering.

Each of the above steps is an implementation of the rationales discussed above in terms of suited CPU-core assignment vs specific performance aspects.

5.2 Binding LPs to Worker Threads

As pointed out earlier, a given set of LPs hosted by K_i gets temporarily bind to a specific worker thread acting within the kernel, which is in charge of performing bottom-half operations related to the LPs in the set, and to schedule them for event processing according to some priority scheme (e.g. Lowest-Timestamp-First). Once the new value for C_i gets defined upon reallocating the computational power, a policy is required to determine which LPs are bind to a specific worker thread. To achieve a binding that allows balancing the whole workload related to local LPs onto the whole set of worker threads, we have devised the below policy. For the j-th LP hosted by kernel K_i, which we refer to as LP_i^j, we compute the total amount of CPU-time required for committing its events during the last observation period (e.g. the last GVT cycle). We refer to this metric as cpu_i^j.

The maximum cpu_i^j value across all the locally hosted LPs represents in our scheme a reference knapsack, and the corresponding LP_i^j is assigned to a given worker thread. Then we exploit the greedy approximation approach proposed by George Dantzig in [Dan57] which allows a maximum "*overflow*" of about 30% over the reference knapsack, in order to build the other knapsacks of LPs (hence knapsacks characterized by sums of cpu_i^x values) to be assigned to the remaining worker threads. We do this by actually applying a variant of the original scheme, where the knapsacks are filled according to a round-robin approach. The procedure is then iterated until no more LP needs to be further bind to any worker thread.

6 Experimental Study

6.1 Test-bed Platform

We have implemented the proposed symmetric multi-threaded optimistic kernel architecture within ROOT-Sim, which is an open source C/MPI-based simulation package targeted at POSIX systems [QPV11a], which implements a general-purpose parallel/distributed simulation environment relying on the optimistic synchronization paradigm.

ROOT-Sim offers a very simple programming model based on the classical notion of simulation-event handlers (both for processing events and for accessing a committed and globally consistent state image upon GVT calculations), to be implemented according to the ANSI-C standard, and transparently supports all the services required to parallelize the execution. It also offers a set of optimized protocols aimed at minimizing the run-time overhead by the platform, thus allowing for high performance and scalability.

Among the main features offered by ROOT-Sim we can mention completely transparent recoverability of the state of the LPs achieved through proper hooking of dynamic memory allocation/release [TQ08], plus ad-hoc code instrumentation schemes that allow incremental determination of dirty state portions [PVQ09] and that, ultimately, allow dynamical switch between different state log/restore schemes depending on the proper dynamics of the application layer [VPQ10].

The single threaded version of ROOT-Sim also offers innovative transparent supports for LP migration and load balancing [PDQ11], which will be considered as a reference for the assessment of the currently presented symmetric multi-threaded version in terms of ability to exploit the computational resources offered by a multi-core machine when the actual simulation workload dynamically varies over time.

Future steps ahead in the development of ROOT-Sim definitely entail the integration of the currently presented symmetric multi-threaded architecture with the aforementioned LP migration subsystem, currently supported only when running in single-threaded mode. This will ultimately provide an environment where the orthogonal capabilities offered by the symmetric multi-threaded paradigm (in terms of dynamic reassignment of the computational power to different kernel instances) and the traditional migration approach (in terms of ability to move individual LPs across different kernel instances) get ultimately combined.

Integration of the multi-threaded approach within ROOT-Sim has been based on `pthread` technology, and on the reorganization of the kernel level data structures in order to (i) provide per-thread private data, and (ii) cache aligned kernel-level memory buffers so to avoid false cache sharing across the worker threads within the same symmetric multi-threaded kernel instance. The latter target has been achieved by exploiting the `posix_memalign` API, plus the usage of proper padding schemes allowing cache alignment for sequences of records, such as arrays of values. As for the accesses to the MPI layer, used to transfer messages across different kernel instances, in our architecture they can be symmetrically issued by any of the worker thread operating within a given kernel instance. Given that the MPI layer does not natively support multi-threading, we have included a wrapper that synchronizes these accesses transparently towards the worker threads via the embedding of critical sections protected by spin-locks.

As far as GVT computation and fossil collection are concerned, we have implemented a symmetric scheme where upon a new GVT computation, all the worker threads operating within a same kernel instance run a race. The race winner actually computes the local reduction and interacts with the master kernel in order to determine the globally reduced value representing the new GVT. However, once defined the new GVT value, all the worker threads operating within the same kernel instance are allowed to perform fossil collection operations in parallel. Each of these threads takes care of fossil collecting the obsolete information associated with its affine LPs.

Finally, the hardware architecture used for testing our proposal is a 64-bit NUMA machine, namely an HP Proliant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 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 is 64-bit Debian 6, with Linux kernel version 2.6.32.5. The compiling and linking tools used are `gcc` 4.4.5 and binutils (`as` and `ld`) 2.20.0.

6.2 Application Benchmarks

In order to evaluate different aspects of the proposed symmetric multi-threaded architecture, we have conducted experiments on two different application benchmarks, namely *PCS (Personal Communication System)* and *Traffic*, which are hereby described. The first one 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 symmetric multi-threaded architecture, while not taking advantages from its ability to reallocate CPU-cores just given the constancy of the workload. The second application benchmark provides instead a highly dynamic workload that varies over time across the involved LPs. This type of benchmark has been used in order to assess the goodness of the symmetric multi-threaded architecture in terms of its ability to reallocate the computational power depending on the actual needs.

6.2.1 The PCS Benchmark

this application benchmark implements a simulation model of wireless communication systems 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 cell. The power regulation model has been implemented according to the results in [KB02].

Upon the start of a call destined to a mobile device currently hosted by a given wireless cell, a call-setup record is instantiated via dynamically-allocated data structures, which gets linked to a list of already active records within that same cell. Each record gets released when the corresponding call ends or is handed-off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call-setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the current call-setup to achieve the threshold-level SIR value. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions (and related variations). The climatic model accounts for variations of the climatic conditions (e.g. the current wind speed) with a minimum time granularity of ten seconds.

This simulation model has been developed for execution on top of ROOT-Sim in a way that each LP models a single wireless cell. Hence, the event-handler callback involves the update of individual cells' states, and cross-LP events are essentially related to hand-offs between different cells.

To evaluate the overhead due to the symmetric multi-threaded architecture, when compared to the classical case of single-threaded optimistic kernel, we have performed a set of experiments where each wireless cell sustains the same workload of incoming calls, hence we are in a balanced scenario not requiring dynamical reallocation of the computational power, which is instead a main target of the symmetric multi-threaded organization. The call inter-arrival time is exponentially distributed, and the average call duration is set to 2 min. The expected rate for call inter-arrival has been set to achieve channel utilization factor on the order of 30%, while the residence time of an active device within a cell has a mean value of 5 min and follows the exponential distribution. For the above scenario, we have run experiments with 1024 wireless cells, modeled as hexagons covering a square region, each one managing 1000 wireless channels.

We have measured the cumulated event rate (expressed as the amount of cumulated committed events vs wall-clock-time) for different configurations of the symmetric multi-threaded kernel, comparing it with the one achievable when running the same ROOT-Sim package in single-threaded mode. In particular, executions with 4, 8, 16 and 32 symmetric multi-threaded kernels (each one starting with 8, 4, 2 and 1 worker thread, respectively) have been carried out. Also, in order to assess the effects of the symmetric multi-threaded organization, we additionally report statistics related to typical run-time parameters characterizing optimistic simulation runs, such as the rollback frequency and the rollback length.

6.2.2 The Traffic Benchmark

this benchmark application simulates a complex highway system (at a single car granularity), where the topology is a generic graph, 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.

At startup phase, the simulation model is asked to distribute the highway's topology on a given number of LPs. Every LP therefore 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.

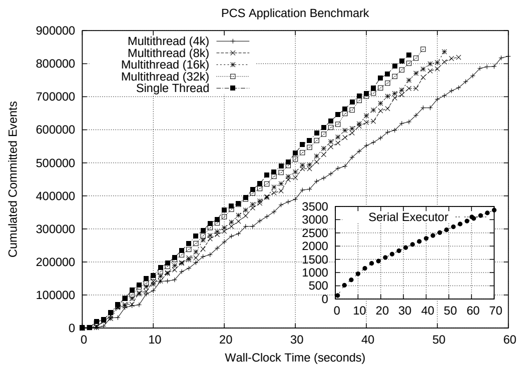
Cars enter the system according to an Erlang probability distribution, with a mean interarrival time specified (for each node) in the topology configuration file. They can join the highway starting from cities/junctions only, and are later directed towards highway segments with a uniform probability. Whenever a car is received, it is enqueued 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. 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.

Accidents are derived according to a probability function as well. In particular, they are more likely to occur when the amount of cars traversing an LP is about half of the cars which can be hosted altogether. In fact, if few cars are in, accidents are less frequent. Similarly, if there are many, the traffic factor produces a speed slowdown, entailing the probability of an accident to occur to be reduced. Therefore, the model discretizes a Normal distribution, computing the Cumulative Density Function in a contour defined as $cars\ in\ the\ node \pm \frac{1}{2}$, having as the mean half of the total number of cars which are at the current moment in the system, and as variance a factor which can be specified at startup. The total number of cars which can be hosted by an LP is computed according to the actual length of the simulated road, which is determined when the model is initialized. When an accident occurs, the cars are not allowed to leave the LP, until the road is freed. The duration of an accident phase is determined according to a Gaussian distribution, the mean and the variance of which are again specified at startup.

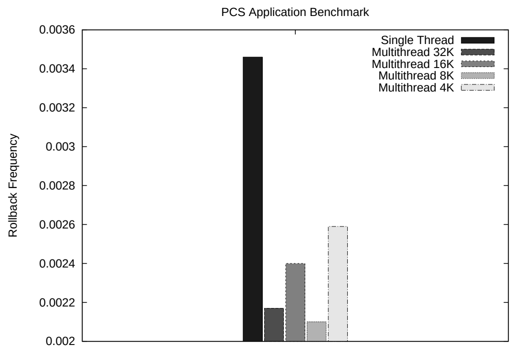
In our execution, we have simulated the whole Italian highway network on top of 1024 LPs. We have discarded the highways segments in the islands in order to simulate an undirected connected graph, which allows to have the actual workload migrating overall the highway. The topology has been derived from [atl], and the traffic parameters have been tuned according to the measurements provided in [tra]. The average speed has been set to 110 Km/h, with a variance of 20 Km/h, and accident durations have been set to 1 hour, with 30 minutes variance. This model has provided results which are statistically close to the real measurements provided in [aci].

We consider this second application benchmark to be significant for showing how our proposed symmetric multi-threaded architecture is able to capture unbalance in the load, and react via computational power reallocation across the active kernels in order to drive the system back into an evenly-distributed workload processing scenario, which would lead to enhanced fruitful exploitation of the computational resources.

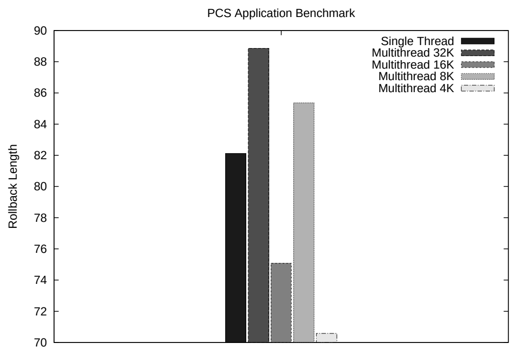
For this benchmark application we still report the event rate, this time comparing it with both the one achieved when considering the classical single-threaded execution mode of ROOT-Sim, and the one achievable when activating within such a single-threaded mode the load balancing mechanisms described in [PDQ11]. We recall again that these load balancing facilities are in principle orthogonal to the facilities offered by the symmetric multi-threaded organization, since their target is the move of LPs across the kernels, not the reassignment of the computational power to the multi-threaded kernels. Anyway, we feel that taking load balancing facilities properly offered by the single-threaded version of ROOT-Sim into account in the comparison provides a relevant reference for assessing the potential offered by the symmetric multi-threaded organization in terms of its ability to fruitfully exploit the available computational power with dynamic workloads.



(a) Cumulated Committed Events



(b) Rollback Frequency



(c) Rollback Length

Figure 3: Results for the PCS Application Benchmark.

6.3 Results

In Figure 3 we show the experimental results that have been obtained for the PCS application benchmark. We recall that this benchmark exhibits balanced workload during the whole run, hence it is suited for assessing the overhead of the symmetric multi-threaded architecture when considering that its capabilities to redistribute the computational power across the different kernel instances are not actually exploited. By the curves related to the cumulated committed events

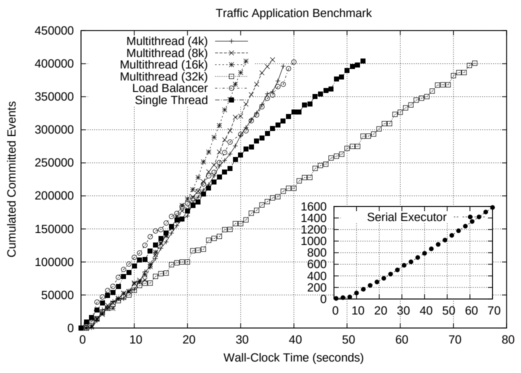


Figure 4: Traffic Benchmark's Throughput

(where all the samples have been obtained as the average over 10 runs all done with different pseudo-random seeds) we see that, unless for the case of 4 multi-threaded kernels (each running 8 worker threads), the additional latency for reaching the completion of the simulation run, compared to the traditional case of single-threaded kernel, is no more than 13%. This is an indication of limited performance intrusiveness by the top/bottom-half architecture while managing the input/output queues of the LPs, as well as limited performance intrusiveness by other synchronization mechanisms (e.g. for the access to the MPI layer), at least when the scale of the multi-threaded configuration is bounded by the value 4. On the other hand, the multi-threaded configuration with 4 kernels and 8 worker threads for each kernel exhibits a non-minimal overhead (on the order of 25%).

We note that, although the workload is constant, small fluctuations due to the probability distribution ruling the generation of the events can arise, which are therefore captured by our symmetric multi-threaded architecture. Nevertheless, this sensibility enhances the reassignment overhead, as long as the time spent in this operation is not rewarded by the new worker threads' configuration. This is more likely to occur exactly when the average number of worker threads per kernel instance gets increased. In addition, we note that these data have been achieved by considering a model with medium-to-fine event granularity, on the order of 30/40 microsecs, thus further supporting the viability of our proposal, since applications exhibiting coarser-grained events would absorb better the actual overhead of the multi-threaded architecture.

Also, we note that the parallel runs provide a super-scalar speedup with respect to the serial executor (based on the calendar-queue scheduler), which indicates that the experimentation has been carried out when considering competitive parallel runs.

For completeness, in Figure 3 we also report the observed values for rollback frequency and rollback length for the PCS application benchmark. By these data we can observe how the symmetric multi-threaded kernel tends to exhibit a slightly throttled execution profile, compared to the single-thread case. In particular, we note a clear reduction of the rollback frequency, with a less significant increase of the rollback length. For the case of the symmetric multi-threaded configuration with 4 kernels, such a throttling is an expression of the above noted overhead, which leads to less favorable run-time dynamics.

In Figure 4 we report the results for the case of the Traffic application benchmark. This time we have compared the cumulated event rate by our symmetric multi-threaded architecture with a classical single-threaded organization, a serial execution of the same application-level software running on top of a calendar-queue scheduler, and also results of the load balancing architecture based on the migration approach presented in [PDQ11]. Again, the parallel approaches provide a super-scalar speedup. The multi-threaded versions of the simulation kernel provide a speedup wrt the single-threaded one, which ranges in between 35% (for the 4 kernels configuration) and 73% (for the 16 kernels configuration).

As for the 32 multi-threaded kernels execution, we note that the speed down is in the order of 37%. This is related to the fact that in this configuration no actual power reallocation is possible on the 32-core server machine that has been employed (in fact, each simulation kernel must have at least one worker thread in order to proceed in the simulation). Therefore, we are again simply measuring the symmetric multi-threaded architecture pure overhead.

The last comparison shown by the plots is the one wrt the traditional load balancer. Although we note that the load balancer configuration provides a speedup in the order of 30% wrt the single-threaded approach, it's throughput is comparable with the 4 kernels multi-threaded configuration, while the 8 and 16 kernels configurations of the multi-threaded architecture are still 30% faster than the traditional load balancer configuration.

7 Conclusions

We have here presented the design and implementation of a symmetric multi-threaded optimistic simulation kernel targeted at multi-core/multi-processor machines, where, similarly to what happens in multi-core oriented Operating Systems in terms of process management, multiple threads operate symmetrically in order to sustain the whole workload associated with the LPs hosted by a kernel instance. This type of organization allows to transparently scale up/down the amount of worker threads operating within a same instance of the optimistic simulation kernel. Hence, it allows for dynamic reassignment of the computational power, namely CPU-cores, to the different kernel instances involved within the optimistic run, depending on variations of the workload associated with the hosted LPs. Policies suited for the reassignment have been also presented, and the whole system has been tested with different application benchmarks.

References

[aci] Aci - dati e statistiche. http://www.aci.it/?id=54.

[atl] Atlante stradale italia. http://www.automap.it/.

[BD97] Azzedine Boukerche and Sajal K. Das. Dynamic load balancing strategies for conservative parallel simulations. In *Proceedings of the 11th ACM/IEEE International Workshop on Parallel and Distributed Simulation (PADS)*, pages 20–28, 1997.

[BYC$^+$05] David Bauer, Garrett Yaun, Christopher D. Carothers, Murat Yuksel, and Shivkumar Kalyanaraman. Seven-o'clock: A new distributed gvt algorithm using network atomic operations. In *In Proceedings of the Workshop on Parallel and Distributed Simulation (PADS'05*, pages 39–48. IEEE Computer Society, 2005.

[CBP00] Christopher D. Carothers, David W. Bauer, and S. Pearce. ROSS: a high performance modular Time Warp system. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pages 53–60. IEEE Computer Society, May 2000.

[CPF99] 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.

[Dan57] George B. Dantzig. Discrete-variable extremum problems. *Operational Research*, (5):–, 1957.

[FH97] R. M. Fujimoto and M. Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446, October 1997.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[GT93] D. W. Glazer and C. Tropper. On process migration and load balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):318–327, 1993.

[HN10] Tsuyoshi Hamada and Keigo Nitadori. 190 tflops astrophysical n-body simulation on a cluster of gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–9. IEEE Computer Society, 2010.

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

[KB02] Sunil Kandukuri and Stephen Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

[lCsLpY$^+$11] Li li Chen, Ya shuai Lu, Yi ping Yao, Shao liang Peng, and Ling da Wu. A well-balanced Time Warp system on multi-core environments. In *Proceedins of the 25th ACM/IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 154–162, 2011.

[LL91] Y. B. Lin and E. D. Lazowska. Processor scheduling for Time Warp parallel simulation. In *Advances in Parallel and Distributed Simulation*, pages 11–14, 1991.

[LW11] Qi Liu and Gabriel Wainer. Multicore acceleration of discrete event system specification systems. *SIMULATION*, 2011.

[Mil10] Ryan James Miller. *Optimistic Parallel Discrete Event Simulation on a Beowulf Cluster of Multi-core Machines.* Master Dissertation, Cicinati University, 2010.

[MMW96] Dale E. Martin, Timothy J. McBrayer, and Philip A. Wilsey. WARPED: A time warp simulation kernel for analysis and application development. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, page 383. IEEE Computer Society, 1996.

[MW95] Larry Mellon and Darrin West. Architectural optimizations to advanced distributed simulation. In *Proceedings of Winter Simulation Conference*, pages 634–641, 1995.

[PDQ11] Sebastiano Peluso, Diego Didona, and Francesco Quaglia. Application transparent migration of simulation objects with generic memory layout. In *Proceedings of the 25th ACM/IEEE International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 169–177. IEEE Computer Society, 2011.

[PF08] A. Park and R. Fujimoto. Optimistic parallel simulation over public resource-computing infrastructures and desktop grids. In *Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 149–156, 2008.

[PLM94] Bruno R. Preiss, Wayne M. Loucks, and Ian D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, July 1994.

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

[QPV11a] F. Quaglia, A. Pellegrini, and R. Vitali. ROOT-Sim: The ROme OpTimistic Simulator: http://www.dis.uniroma1.it/∼hpdcs/root-sim/, October 2011.

[QPV11b] Francesco Quaglia, Alessandro Pellegrini, and Roberto Vitali. ROOT-Sim: The ROme OpTimistic Simulator: http://www.dis.uniroma1.it/∼hpdcs/root-sim/, October 2011.

[QS03] Francesco Quaglia and Andrea Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, June 2003.

[Qua01] Francesco Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, February 2001.

[RA94] Robert Rönngren and Rassul Ayani. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 110–117. Society for Computer Simulation, July 1994.

[RA97] Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, 1997.

[RLAM96] Robert Rönngren, Michael Liljenstam, Rassul Ayani, and Johan Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 70–77. IEEE Computer Society, May 1996.

[SQ10] Tiziano Santoro and Francesco Quaglia. A low-overhead constant-time ltf scheduler for optimistic simulation systems. In *IEEE Symposium on Computers and Communications (ISCC)*, page 948, 2010.

[TQ08] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the 22nd ACM/IEEE International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 163–172. IEEE Computer Society, 2008.

[tra] http://www.autostrade.it/studi/studi_traffico.html.

[VPQ10] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Proceedings of the 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 319–327. IEEE Computer Society, 2010.