Towards a Fully Non-blocking Share-everything PDES Platform

Mauro Ianni, Romolo Marotta, Alessandro Pellegrini, Francesco Quaglia
DIAG—Sapienza Università di Roma
{*mianni,marotta,pellegrini,quaglia*}*@dis.uniroma1.it*

***Abstract*—Shared-memory multi-core platforms are changing the nature of Parallel Discrete Event Simulation (PDES) because of the possibility to fully share the workload of events to be processed across threads. In this context, one rising PDES paradigm—referred to as share-everything PDES—is no longer based on the concept of (temporary) biding of simulation objects to worker threads. Rather, each worker threads can—at any time—pick from a fully shared event pool an event to process which can be destined to whatever simulation object. While attention has been posed on the design of concurrent shared pools, allowing non-blocking parallel operations, the scenario where two (or more) threads pick events destined to the same simulation object still lacks adequate synchronization support. In fact, these events are currently sequentialized and processes in a critical section touching the simulation object state, thus leading threads to mutually block each other. In this article we present the design of a share-everything speculative PDES engine that prevents mutual thread blocks because of the access to a same object state. In our design, the non-blocking property is seen as a vertical attribute of the engine (not only of the event pool). This vertical view demands for both innovative event-dispatching schemes and, at the same time, innovative interactions with (and management of) the fully-shared event pool, which are features that we embed in our innovative design.**

I. Introduction

Complex and/or large simulation models demand parallelization techniques for making model-execution feasible. These are built on top of consolidated methodologies. However, along time, they are also influenced by raising trends in the hardware of underlying computing platforms.

For the case of discrete event simulation, classical parallelization methodologies are based on dividing a complex model in multiple simulation objects that interact via cross-exchange of timestamped events [1]. Also, the implementation of Parallel Discrete Event Simulation (PDES) platforms adhering to these methodologies has been historically based on explicitly assigning groups of simulation objects to different worker threads. Periodic rebinding of objects to threads has been actuated in order to keep resource usage well balanced on the medium/long term (see, e.g., [2], [3]) according to a classical view where the workload is migrated towards the available computing power.

In recent years, new implementation trends for PDES platforms have arisen, which have been based on the new concept of migrating the computing power towards the workload. This has been possible thanks to the advent and large diffusion of shared-memory multi-core machines, which give the possibility to fully-share the state of the simulation objects—as well as platform level data structures such as the event pool–across all the concurrent worker threads operating within the PDES environment.

More in detail, the recent *share-everything* PDES paradigm [4], [5] has been based on the concept of a unique and fully-shared event pool, keeping the unprocessed events destined to whatever simulation object, from which any worker thread picks its next event to process. This allows delivering the computing power to the highest priority events—those with lower timestamps that are currently kept within the shared event pool—along the whole lifetime of the simulation. This is important since it can favor the prompt advancement of the model execution independently of whether conservative or speculative event-processing is adopted. Such a capability is exactly originated by the fact that a thread is not bound to any specific simulation object, rather it can switch between different objects on a per-event basis.

The downside of this organization is related to the need for thread-level coordination mechanisms, which are required in order to correctly handle the access to the fully shared set of data structures implementing either the simulation engine internal logic or the application level one. As for the engine level, recent results have provided highly concurrent fully shared-event pools supporting non-blocking operations [4], [6]. They enable scalability of engine level tasks, in particular by preventing extraction and insertion operations from/to the event pool to become a bottleneck.

However, at current date, if two or more threads concurrently pick different events destined to the same simulation object, they incur the risk of blocks and sequentialization. In fact, the application code is typically designed in order to have an object representing a sequential entity, on whose state image a single thread is enabled to work at any given point in time in order not to impair safety of software actions because of concurrent conflicting accesses. Also, the block is usually implemented via spin-locks in order to avoid operating system thread-reschedule delays caused by kernel-level blocking synchronization services. As a consequence, an additional penalty comes out from the energy-waste generated by the corresponding busy-waiting CPU cycles.

In order to cope with these shortcomings, in this article we present the design of a share-everything PDES platform

where no thread is ever blocked because of a conflicting concurrent access to some engine/application data structure. This result has been achieved by fully revising the CPU-dispatching rules that are put in place by worker threads, which are in our proposal no longer based on extracting the element with the current minimum timestamp from the fully-shared event pool (as an individual action) and then on locking the destination object (as a final action for CPU-dispatching the event). Rather, the pool is accessed in non-blocking fashion by selecting for processing an event destined to a simulation object that is currently not active—not already CPU-dispatched by any other worker thread. This is discovered exactly while traversing the event pool, still in non-blocking fashion, thanks to the introduction of new signaling mechanisms based on metadata that are used to indicate the state of each object (already CPU-dispatched or not), which are manipulated in combination with the access to the record representing an element of the event pool. These metadata are manipulated atomically in non-blocking mode via the Compare&Swap (CAS) machine instruction.

Overall, our proposal for the achievement of a non-blocking share-everything PDES engine relies on a vertical approach where the state of the event pool, in combination with the augmented metadata, does not only keep into account what events need to occur at a given object. Rather it also expresses the current state of the object, namely whether it is already CPU-dispatched or not, and the whole information is accessed/manipulated in non-blocking mode.

Our current design is towards a fully non-blocking speculative PDES engine, where the actual need for thread blocking is only determined by the setup level of speculation. In our current design we set this level to one for each object—thus each object is allowed to perform a step ahead in logical time speculatively—but nothing prevents the possibility to integrate more complex state restore mechanisms that would enable speculating along chains of events. This facility can be seen as orthogonal to the core CPU-dispatching mechanisms of events that we present in this article. In any case, even though we target speculative event processing, we are still able to exploit lookahead information in order to detect wether some event is safe to process and does not require reversibility support. This allows us to enable simulation objects to perform as many steps a head as possible in non-speculative mode before trapping into speculative execution of some event.

We present experimental results achieved by running our solution on top of a 32-core HP ProLiant server. As case study applications, we have run several configurations of the classical PHOLD benchmark [20] and compared the performance achievable by our current proposal with the one delivered by the share-everything PDES engine recently presented in [6].

The remainder of this article is structured as follows. In Section II we discuss related work. Section III provides the description of our non-blocking speculative PDES platform. Experimental data are presented in Section IV.

II. Related Work

Our proposal is related to the literature approaches that try to exploit shared-memory multi-core machines in order to optimize the runtime behavior of PDES engines. A few results in this area have been targeted at traditional PDES platform organizations, based on explicit partitioning of the workloads of simulation objects across the different worker threads. The works in [7]–[9] address issues related to the optimization of the message-passing architecture. The solution in [10] tackles the problem of runtime optimization of the binding between simulation objects and worker threads. The recent proposal in [11] copes with the issue of porting traditional PDES paradigms, such as Time Warp [12], onto multi-core GPU devices. Compared to all these approaches, we tackle the different case where simulation objects are not partitioned across threads. Rather the threads can switch across different objects in a per-event basis.

Considering that our PDES platform leads multiple worker threads to share accesses to the same simulation object by different threads in non-blocking mode, our proposal is also related to the one in [13]. Here the authors explore how to use non-blocking Software Transactional Memory support in order to manipulate object attributes concurrently. However, the sharing of attribute accesses is confined to specific attributes that need to be identified at compile/link time and need to be explicitly managed via STM APIs. Our solution is instead fully transparent to the application code. Also, in our non-blocking engine we do not use techniques to reconcile conflicting concurrent actions (in a commit or a rollback). Rather, we avoid at all the conflicts thanks to the smart CPU-dispatching scheme of events that exploits non-blocking accesses to the underlying event pool in order to select the object to be taken in charge by a worker thread.

The work in [14] introduces the concept of cross-state-events in PDES, which are events that can touch the state of multiple objects. With this scheme, two or more worker threads can exhibit shared accesses to a same simulation objects, since the boundaries of the activities of the threads are no longer defined on the basis of the partitioning of the model into disjoint object states. Rather, a cross-state-event leads such boundary to be broken. However, this solution is still based on a preliminary binding of objects to threads, which we avoid in our share-everything solution.

Clearly, our proposal is also related to all the works that try to exploit non-blocking algorithms in order to optimize the performance of PDES systems. Most of these approaches have been tailored to the management of engine level data structures, such as the event pools [4], [15], [16]. On the contrary, we include management rules that lead threads to never block each other even in scenarios of full share of

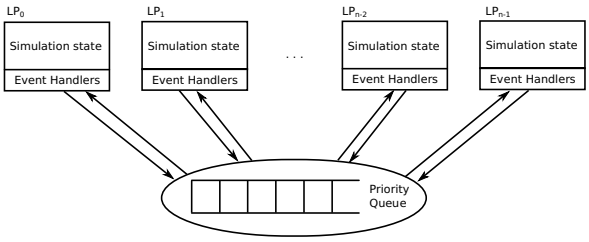

Figure 1. Basic engine organization.

the objects' states. Also, a few of them—such as [15]—are oriented to scenarios with workload partitioned across the worker threads while we cope with fully-shared workload.

III. The PDES Platform

A. Basics

Our target is a share-everything PDES platform whose general organization is provided in Figure 1. This organization conforms to the archetypal one that has been presented in [4]. Similarly to the traditional PDES paradigm, the share-everything paradigm still supports the partitioning of the simulation model in multiple simulation objects—classically referred to as Logical Processes (LP)—each one associated with a unique identifier $n \in [0, N-1]$ and whose state is kept in a memory region that is disjoint from the regions keeping the other LPs' states.

To support parallel execution, the platform runs multiple worker threads (WTs). Each WT can extract events from a fully-shared pool. Once picked an event, the corresponding event handler, namely the one targeting the destination LP, is CPU-dispatched. A WT extracts from the shared pool an event that currently stands at higher priority level—e.g., the unprocessed event with the lowest timestamp. This allows concentrating the computing power offered by the overall set of WTs close to the commit horizon of the simulation, thus reducing the likelihood that some LP can run ahead in simulation time while others remain back. For the case of speculative processing, which is the target in our design, this allows reducing the likelihood of erroneous speculation and tends to improve the efficiency of the parallel run.

Each LP has its own event handler, although multiple LPs can be governed by a same event handler routine. This module is in charge of taking over an event and processing it. The processing of the event possibly produces state updates and generates new events that can be in principle destined to any LP involved in the simulation. These events transit into the shared-event pool and the delivery of an event to the correct destination LP is demanded to the simulation engine. In fact, upon extracting an event from the shared pool, the WT checks with the metadata that are included in the event envelope and determines to what LP the event needs to be routed for processing.

With this organization, the PDES platform is in charge of two core aspects:

- It needs to generate an evolution of the state of each LP that is causally consistent, namely no timestamp inversion must figure out as ever occurring for events processed at the LP;
- It needs to guarantee correctness of the accesses by WTs to the overall set of shared data structures representing the state of the engine and of the LPs.

In our current design we enable an LP to process at most one event speculatively, while it can process as much events are available from the shared pool if they are safe—meaning that their processing will never violate causality. If an event is processed speculatively, then the WT processing it needs to eventually determine if it can be committed or needs to be rolled back. In the latter case, any new event possibly produced by the event processing phase is simply discarded, otherwise it is inserted into the shared event pool. This means that new events produced by the execution of one event e are kept in a private WT space and flushed to the globally shared event pool only upon determining the commitment of e, if any.

The sorting of the events within the shared pool is based on event timestamps, and each event-envelope also entails information about the destination LP. Moreover, for what asserted before in relation to the management of speculative vs safe processing, all the events kept in the event pool are *schedule-committed*, namely they will never need to be retracted because of a rollback leading to undo speculatively processe events.

Recent literature proposals in the area of share-everything PDES rely on non-blocking event pool operations for scalability, but employ blocking accesses to the states of the LPs. The blocks, typically implemented via spin-locks, are used to guarantee isolation of the state snapshot manipulations by the concurrent WTs that need to operate on a same LP. In our design we aim at avoiding that two or more WTs block one another because of the need for accessing the same LP state, and we still relying on non-blocking event pool operations. In particular, we enable WTs to always work on disjoint portions the application state, namely different LPs, while still concentrating their work on the higher priority events kept by the shared pool. This leads to the avoidance for blocking phases, with consequent save of CPU-time that can be devoted to actual event processing job.

To reach this goal, our event pool is coupled with information related to the current CPU-dispatch of the LPs among WTs. In other words the pool is associated with information related to a kind of *short-term* binding of LPs to WTs. In more details, we introduce the concept of booking of LPs by WTs, so that a non-blocking dequeue operation by some WT that tries to CPU-dispatch an event destined to some LP does not actually remove the corresponding event-record item from the shared pool, rather it simply "books" the target LP, thus putting in place the aforementioned short-term binding. In our implementation the booking is based

on executing a CAS machine instruction that implements a non-blocking try-lock operation on a variable associate with the LP to be booked.

The event records are not removed when the corresponding LP is booked and they are being processed, since as we shall explain their presence in the event pool is exploited for detecting the safety of the events themselves. So the booking of an LP figures out as a logical removal of some corresponding event, which is not yet finalized into a definite removal. The latter takes place when event processing is committed. Consequently the event pool data-structure has an additional API to effectively remove some item from the event pool, so as to perform garbage collection of event records whenever they are no longer needed—since their processing is related to a definitive committed action.

Another capability offered by our non-blocking event pool at the core of the fully non-blocking PDES engine is the one of enabling the monotonic navigation of the pending event set starting from the minimum—denoted as *head*. This capability can be easily implemented on any priority queue based on linear arrangement of the elements (tree-based event pools might require $O(\log n)$ steps for each step forward). In our solution we have chosen a lock-free Calendar Queue [4], [17] that guarantees $O(1)$ complexity to perform enqueue/dequeue operations in a concurrent scenario, appropriately integrated with the simulation engine itself via the aforementioned LP booking mechanism.

The procedure of real node deletion can be easily implemented in $O(1)$, since the caller WT already knows the "position" of the target node in memory. In fact, a node can be eventually removed only after the associated LP has been booked by a WT and the event itself has been processed with assurance of not violating causality—so it will no longer serve for additional processing.

To manage recoverabilty of the LPs' states, which is needed if speculation leads to violate causality, we rely on a transparent *static binary instrumentation* technique of event handlers. In more detail, we exploit the Hijacker [18] open-source customizable static binary instrumentation tool, tailored to the ELF format, which is able (before the final linking stage of the application-level simulation model) to alter a program's execution flow. In particular, according to some user-specified rules, it identifies any memory writing instruction and places before it a call to a module which reads the memory location before its update in order to generate an instruction able to undo the corresponding effect according to the proposal in [19].

In speculative PDES the evolution of the simulation is characterized by the progress of the *commit horizon*, commonly referred to as Global Virtual Time (GVT), which represents a break point along the time axis that divides events which might be still undone, due to causality violations, by events which will never been undone, namely committed events. Since the execution of a simulation event

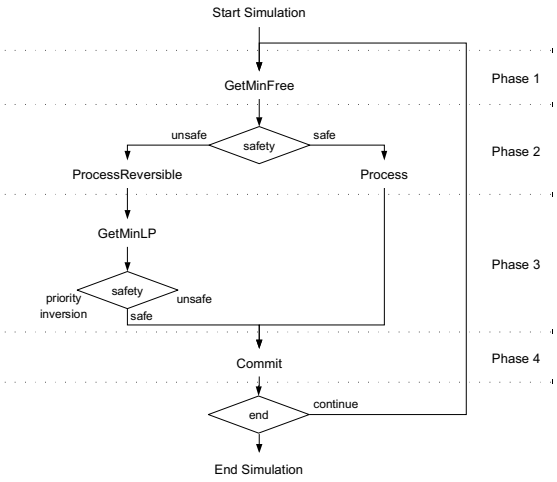

Figure 2. diagram of the simulation loop

Algorithm 1 Main loop

```
 1: procedure MAINLOOP( )
 2:     Set<event> newEvents ← ∅
 3:     bool safe ← FALSE
 4:     event e, e' ← NULL
 5:     while ¬endSimulation do
 6:         ⟨e, safe⟩ ← GETMINFREE( )
 7:         if e = NULL then
 8:             continue
 9: execute:
10:         if safe then
11:             newEvents ← PROCESS(e)
12:         else
13:             newEvents ← PROCESSREVERSIBLE(e)
14:             do
15:                 ⟨e', safe⟩ ← GETMINLP(LP(e))
16:                 if e ≠ e' then
17:                     e ← e'
18:                     newEvents ← ∅
19:                     goto execute
20:             while ¬safe ∧ ¬endSimulation
21:         if endSimulation then
22:             break
23:         COMMIT(e, newEvents)
```

e at simulation time T can only generate some event e' with a timestamp $T' \geq T$, which means that an event can not affect the past, the GVT can be identified as the smallest timestamp across all the unprocessed/uncommitted events in the system. At any time, the commit horizon event can be considered as a *safe* (namely, causally consistent) one, and therefore, if not yet processed, does not require any reversibility mechanism for its execution. This means that the corresponding WT can run a non-instrumented version of the target event handler which installs the updates on the LP state with no possibility to eventually undo them

B. Platform Architecture

The pseudo-code of the main loop of our simulation engine, which is executed by all WTs, is shown in Algorithm 1. The loop can be logically divided in four different phases,

as shown in Figure 2. Initially a call to the GetMinFree procedure is executed in order to retrieve from the shared event pool an event to process. As the name of the procedure suggests, in this phase we try to pick an event for processing destined to some "free" LP—say not currently CPU-dispatched. Also, the even should have the minimum possible timestamp, although it might not coincide with the absolute minimum in the event pool.

The pseudo-code for the GetMinFree procedure is shown in Algorithm 2. This procedure traverses the event pool starting from the head. At each step of traversing of some event, the WT tries to book the corresponding destination LP. Operatively the WT tries to lock the LP via a non-blocking CAS machine instruction.

If booking fails it means that some other WT is already working on the LP. In this case, the WT continues traversing the event pool for a new try with some subsequent event. When an event whose corresponding LP can be booked is found, the event is returned, although it is not definitely removed from the pool. Moreover, while searching for an event to take care of, the procedure is able to determine the safety of such event. In more details, the procedure keeps track of the LPs met depending on the traversed events while scanning the pool, which where already found to be booked, and of the timestamps of the corresponding events. Considering the possibility of having a lookahead specification for the simulation model, an event can be returned as safe for processing if the difference between its timestamp and the minimum one still recorded into the event pool (the GVT) is smaller than the lookahead value. If this is true, it means that no other event will ever be delivered—because of pending processing activities at any LP—to the same LP targeted by the event in its past. On the other hand, concurrent processing of multiple events at a same LP is prohibited because of the booking mechanism, which leads to safety of processing—namely WT isolated processing on a given LP—even in scenarios where an event destined to the same LP will be successively flushed to the event pool with a timestamp lower than the currently selected event. This situation, if materialized, will be resolved via rollback, as we shall discuss. In any case the check in Line 8 also covers the scenario where the booked LP corresponds to one that was previously attempted to book and that has been in the meanwhile released by some other WT. If this scenario materializes, the picked event is not safe independently of the lookahead since there is another event to process at the sane LP that stands in the past of the picked one.

Once the GetMinFree procedure ends, it returns an event associated with an LP that is univocally bound to the WT, together with the indication of whether the event is safe to process or needs to be processed speculatively. Note again that the event is still available in the queue. On the other hand, it can be considered as *logically extracted* since, until the WT will keep the lock on the corresponding LP, no other

Algorithm 2 GetMinFree operation

```
1: procedure GETMINFREE( )
2:     Set S ← NULL
3:     node n ← event_pool.min()
4:     time min ← n.ts
5:     while (¬TRYLOCK(n.lp) do
6:         S.add(n.lp)
7:         n ← event_pool.next(n)
8:     if n.ts < min + LOOKAHEAD ∧ ¬n.lp ∈ S then
9:         return ⟨n.event, TRUE ⟩
10:    else
11:        return ⟨n.event, FALSE ⟩
```

Algorithm 3 GetMinLp operation

```
1: procedure GETMINLP(int lp)
2:     node n ← event_pool.min()
3:     time min ← n.ts
4:     while (n.lp ≠ lp) do
5:         n ← event_pool.next(n)
6:     if n.ts < min+ LOOKAHEAD then
7:         return ⟨n.event, TRUE ⟩
8:     else
9:         return ⟨n.event, FALSE ⟩
```

WT will extract events destined to the same LP. At this point, according to the *safe* value returned by the GetMinFree procedure, two different execution paths are possible: safe and unsafe.

A safe execution leads the WT to CPU-dispatch the non-instrumented event handler, which instals the updates on the LP state in non-reversible mode. Instead, for an unsafe execution, we dispatch the instrumented version, which entails undoing capabilities. At the end of the speculative execution of an event e, in order to preserve causal consistency and schedule-commitment for all the events inserted into the shared pool, the WT has to wait that the executed event becomes safe. This check is done via the GetMinLP procedure, whose pseudo-code is shown in Algorithm 3. The execution path of the GetMinLP procedure is somehow similar to the one of the GetMinFree procedure. As the name suggests, this time we are not looking for a generic event within the pool, but rather we are trying to discover what is the minimum timestamp event destined to the very same LP that was targeted by the event e, in the hope such an event corresponds to e and has become safe in the meanwhile.

In this procedure, the head of the pool is retrieved and the pool is traverse searching for the first event targeted at the same LP of the event e. Once found, the event is returned together with its safety condition, this time based only on the distance from the commit horizon and the lookahead. It is important to note that, we have no guarantee on which event is fetched this time, in fact, if the procedure returns an event different from e it means that a priority inversion has occurred—event e has been executed out of timestamp order. In this case, the processed event e is rolled-back executing undo-code blocks generated by instrumented handlers and

Algorithm 4 Commit operation

```
1: procedure COMMIT(event e, Set<event> E)
2:     ∀e′ ∈ E: INSERTINEVENTPOOL(e′)
3:     DELETE(e)
4:     UNLOCK(LP(e))
```

the loop is restarted with the newly extracted event. If there is no priority inversion, the `GetMinLP` procedure is repeatedly called until the processed event is returned as safe. When this condition occurs, the event is finalized towards its commitment by proceeding to the next phase of the main loop in Algorithm 1.

The pseudo-code of the `commit` procedure is shown in Algorithm 4. The `commit` procedure first places into the global pool all the newly produced events and successively eliminates, the just committed event e making the commit horizon progress to the next event. Once the operations on the pool are completed, the LP's lock is released and the loop is restarted checking each time if the simulation is completed.

We remark again that all the operations on the event pool (insertions/traversals/deletions) are implemented in non-blocking mode according to the solutions presented in [4], [17], which guarantee scalability of concurrent event pool manipulation. Clearly, the smart dispatching mechanism we introduce based on LPs' booking adds a new dimension in terms of scalability by avoiding WTs' blocks for accessing the LPs' states in scenarios with full sharing of the LPs and of their workload.

C. Optimization

1) Reserving Nodes: The `GetMinFree` operation is one of the most onerous in our PDES engine. In order to reduce the number of atomic operations, namely CAS, to be performed to support the booking mechanism, a node has been augmented with a *reserved* field. This is updated by a classical write once acquired the relative lock on (once booked) the corresponding LP. This field is not required for correctness, therefore if an update is lost—given that it is not performed using atomic operations—no problem actually arises, since the WT will fined the corresponding LP already booked even in scenarios of false negatives for node reserving. On the other hand, if a WT finds a node reserved, it will not access the lock variable of the corresponding LP via CAS, thus avoiding the cost of a memory (remote) access and possibly the cost of the corresponding atomic instruction performing the access.

IV. Experimental Results

We have integrated our proposal in a new platform release of our open source share-everything PDES project ([1]). In this section we report a performance comparison of this proposal vs the last release of that same engine as presented in [6] which was based on non-blocking event pool operations, but on blocking access to the LPs' states.

As test-bed application we used the classical PHOLD benchmark [20] configured with 1024 LPs. Each LP schedules events for any other LP in the system, with an exponential timestamp increment. As usual for PHOLD, event processing leads to spending some CPU time, via a busy loop emulating a given event granularity. In our experiments we initially set the loop to give rise to events with granularity of the order of 60 microseconds, which can be considered as a mid-weight value.

In our PHOLD configuration we included 10 hot spot LPs, towards which a given percentage of events injected by the other LPs are routed. This percentage has been varied from 25% to 100%, passing through 50% an 75%. On the other hand, when an event is processed at an hot spot, the newly generated event by the processing stage is destined to whichever LP. It is known that PDES workloads with hot spots are difficult to manage since they might provide unbalance in case of traditional PDES platforms relying in the binding between LPs and WTs. Also, they are difficult to manage in share-everything PDES systems where WTs can block one another because of the need to process events on a same LP (the hot spot one). We decided to experiment with this kind of complex workload just to study how our new approach could overcome such known limitations.

All the tests have be run on a 32-core HP ProLiant machine running Linux (kernel 3.2) equipped with 64 GB of RAM. The number of WTs running within the PDES platform has been varied from 1 to 32, just in order to perform a scalability study. All the reported data points have been computed as the average over 10 runs, executed with different seeds for the pseudo-random generation of event timestamps.

As a final preliminary note, we set the lookahead of the PHOLD model to the 10% of the average timestamp increment of newly generated events. This enabled us to study the effects of our non-blocking approach in scenarios where the need to waiting for the safety of speculatively processed events does not become the major limiting factor to scalability.

In Figure 3 we show results related to the speedup observed while varying the number of WTs for the different configurations of the PHOLD model with hot spots. Speedup results have been computed over a sequential run of the same PHOLD model carrie out on a classical calendar-queue scheduler. The two reported plots in each graph refer to our new non-blocking share-everything PDES engine (NBSE) and to the one in [6], which is based on LPs' states locks (SLSE) with sequentialization of WTs' conflicting accesses to the same LP state. By the plots we see how our NBSE proposal is definitely resilient to performance degradation while increasing the number of used WTs. In fact, its speedup does not decrease for larger thread counts except for the scenario

[1] Available at `https://github.com/HPDCS`

where the hot spots are hit with probability 1.0. In any case, even for such extremely skewed workload of events, the decrease of the speedup when moving from 24 to 32 WTs is minor. Instead, the SLSE configuration shows a worse scalability, which leads performance to definitely decrease, especially when considering higher values of the probability to hit the hot spot LPs with newly generated events. The peek speedup achieved by NBSE is around 15 for all the configurations, while for largely skewed accesses SLSE does not provide more than 10 as the speedup. This is a relevant achievement of NBSE, showing not only scalability, but rather the capability to maintain similar scalability levels independently of the actual pattern of events (more or less clustered) across the LPs.

To further support the effectiveness of NBSE, we report results related to a few additional variations of the PHOLD configuration. In a first variation, we reduce the event granularity from 60 to 40 microseconds. In principle, these settings should be favorable to SLSE since the ratio between the time spent in non-blocking event pool operations and the one spent processing events is increased. Hence, SLSE should suffer a bit less from the need for executing sequentialized accesses to the LPs' states while processing events. In any case, by the results in Figure 4 we se how NBSE still remains definitely more performing than SLSE. In fact, NBSE still guarantees the order of 15 of speedup, while SLSE does not offer more than 10 of speedup. These data refer to the cases of medium (0.5) and high (0.75) probability of event routing towards the host spots.

The additional variation of PHOLD we consider is even more favorable to SLSE, since it is based on including 32 hot spots (which corresponds to the maximum number of WTs that are run within the PDES environment), rather than only 10. With these settings, the workload of events is less skewed, in terms if its distribution across the LPs, so that SLSE can find more opportunities of no-conflict between WTs that need to process events at the same destination LP. Hence, it should suffer less from lock-based sequentialization of the accesses to a same LP's state. The results for this variation of PHOLD are reported in Figure 5. Although the data show that SLSE suffers less from reduced scalability while increasing the number of WTs, the plots still show how NBSE is superior, especially with higher likelihood of hitting the spots. In fact, the maximum speedup achieve with NBSE is of the order of 16, while SLSE does not provide more that 12.5 speedup. Overall, also for this more favorable workload to SLSE, NBSE still allows up to almost 30% better speedup values.

V. Conclusions and Future Work

In this article we have presented a PDES engine, suited for shared-memory multi-core machines, which is designed according to the share-everything paradigm. With this approach, the worker threads fully share the workload of events that are kept by a unique (fully shared) event pool. The core problem we addressed has been the one of avoiding that two or more threads pick from the shared-event pool events destined to the same simulation object. In fact, these would require sequentialization of the actions by the threads, given that each object in a PDES run is an intrinsically sequential entity. We achieve this goal by still having the computing power offered by the worker threads concentrated on higher priority (lower timestamp) events. As a matter of fact our approach slides towards a fully non-blocking share-everything PDES engine since we avoid threads blocks at the simulation object level and also exploit non-blocking event pool algorithms. This enables our proposal to show excellent scalability even in scenarios with skewed workloads across the simulation objects. We included in our PDES engine design the possibility to process events speculatively. However, at current data we enable at most one speculative step forward for each simulation object. This limited speculation level is the only blocking element within our engine. Our plan for future work is to remove this limitation, which would in principle enable a completely non-blocking share-everything PDES platform fully unleashing non-blocking management of share-data data—as we currently do—and fully speculative/non-blocking virtual time synchronization.

References

[1] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[2] C. D. Carothers and R. Fujimoto, "Efficient execution of time warp programs on heterogeneous, NOW platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 3, pp. 299–317, 2000.

[3] S. Peluso, D. Didona, and F. Quaglia, "Supports for transparent object-migration in PDES systems," *J. Simulation*, vol. 6, no. 4, pp. 279–293, 2012.

[4] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A lock-free o(1) event pool and its application to share-everything pdes platforms," in *Proceedings of the 20th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, ser. DS-RT. IEEE Computer Society, Sep. 2016, winner of the Best Paper Award.

[5] E. Santini, M. Ianni, A. Pellegrini, and F. Quaglia, "Hardware-transactional-memory based speculative parallel discrete event simulation of very fine grain models," in *Proceedings of the 22nd International Conference on High Performance Computing*, ser. HiPC. IEEE, dec 2015, pp. 145–154.

[6] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A conflict-resilient lock-free calendar queue for scalable share-everything PDES platforms," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2017, Singapore, May 24-26, 2017*, 2017, pp. 15–26.

[7] B. P. Swenson and G. F. Riley, "A New Approach to Zero-Copy Message Passing with Reversible Memory Allocation in Multi-core Architectures." in *PADS*, 2012, pp. 44–52.

[8] R. Vitali, A. Pellegrini, and F. Quaglia, "Towards symmetric multi-threaded optimistic simulation kernels," in *26th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation, PADS 2012, Zhangjiajie, China, July 15-19, 2012*, 2012, pp. 211–220.

[9] J. Wang, D. Jagtap, N. B. Abu-Ghazaleh, and D. Ponomarev, "Parallel discrete event simulation for multi-core systems: Analysis and optimization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1574–1584, 2014.

[10] R. Vitali, A. Pellegrini, and F. Quaglia, "Load sharing for optimistic parallel simulations on multi core machines," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 2–11, jan 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2425248.2425250

[11] X. Liu and P. Andelfinger, "Time warp on the GPU: design and assessment," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2017, Singapore, May 24-26, 2017*, 2017, pp. 109–120.

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

[13] L.-l. Chen, Y.-s. Lu, Y.-P. Yao, S.-l. Peng, and L.-d. Wu, "A Well-Balanced Time Warp System on Multi-Core Environments," in *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS. IEEE Computer Society, 2011, pp. 1–9.

[14] A. Pellegrini and F. Quaglia, "Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies," in *Proceedings of the 2014 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. PADS. ACM Press, 2014, pp. 105–116. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2601381.2601398

[15] J. Hay and P. A. Wilsey, "Experiments with hardware-based transactional memory in parallel simulation," in *Proceedings of the 2015 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. PADS. ACM Press, 2015, pp. 75–86.

[16] S. Gupta and P. A. Wilsey, "Lock-free pending event set management in Time Warp," in *Proceedings of the 2014 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. PADS. ACM Press, 2014, pp. 15–26.

[17] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A Non-Blocking Priority Queue for the Pending Event Set," in *Proceedings of the 9th ICST Conference of Simulation Tools and Techniques*, ser. SIMUTools. ICST, 2016.

[18] A. Pellegrini, "Hijacker: Efficient static software instrumentation with applications in high performance computing," in *Proceedings of the 2013 International Conference on High Performance Computing and Simulation*, ser. HPCS, Helsinki, Finland, 2013, pp. 650–655.

[19] D. Cingolani, A. Pellegrini, and F. Quaglia, "Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES," in *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. PADS. ACM Press, 2015.

[20] R. M. Fujimoto, "Performance of Time Warp Under Synthetic Workloads," in *Proceedings of the Multiconference on Distributed Simulation*. Society for Computer Simulation, 1990, pp. 23–28.

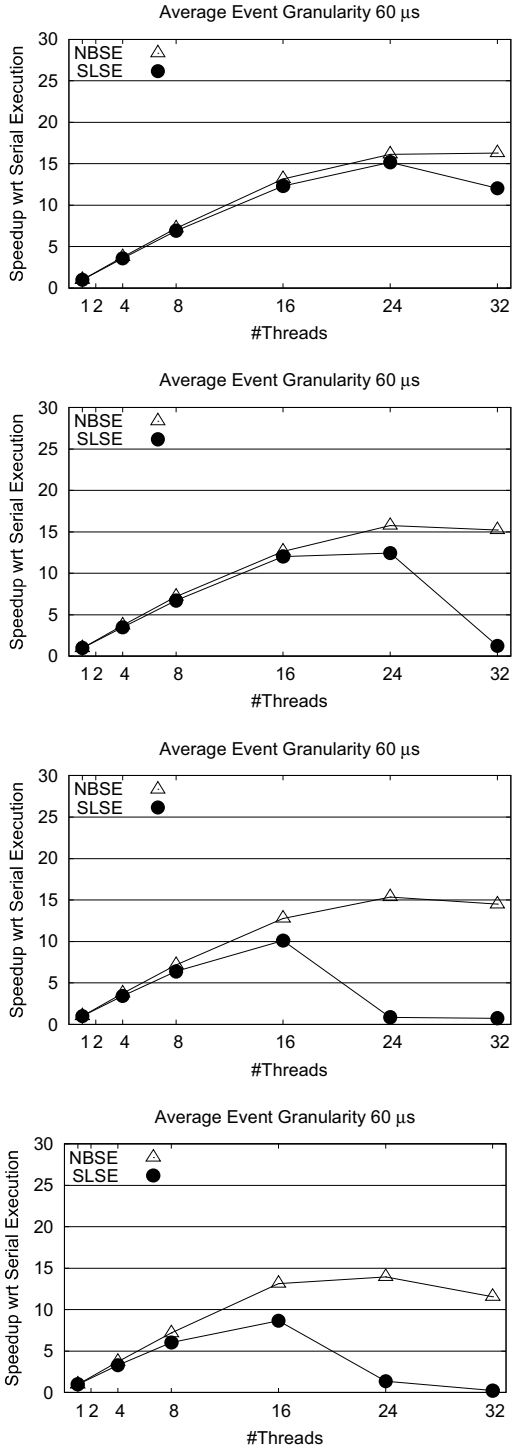

Figure 3. Results with PHOLD - hot spots configurations - the probability of hitting the spot increases from 0.25 to 0.5, 0.75 and 1 from the top graph to the bottom graph.

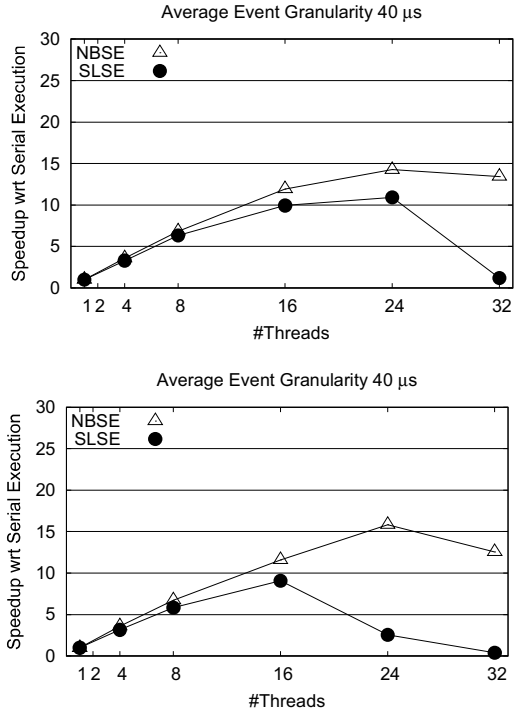

Figure 4. Results with PHOLD (variation with event granularity set to 40 microseconds) - the probability of hitting the spots increases from 0.5 to 0.75 from the top graph to the bottom graph.

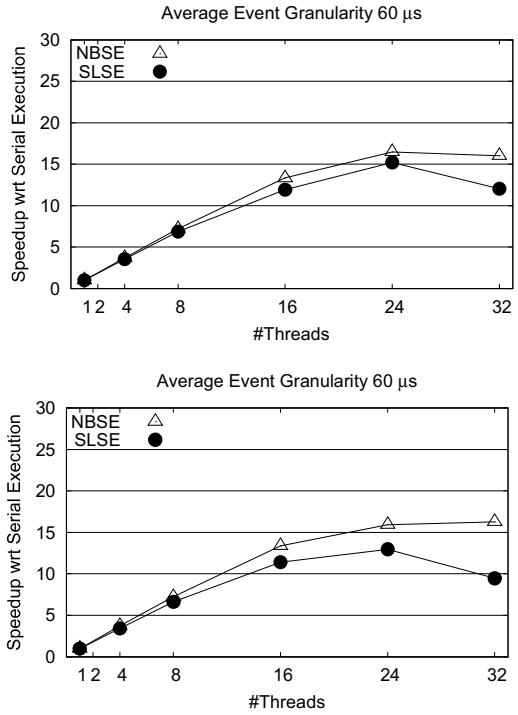

Figure 5. Results with PHOLD (variation with 32 hot spots) - the probability of hitting the spots increases from 0.5 to 1.0 from the top graph to the bottom graph.