The Ultimate Share-Everything PDES System

Mauro Ianni
Sapienza, University of Rome
Lockless S.r.l.
mianni@diag.uniroma1.it
ianni@lockless.it

Romolo Marotta
Sapienza, University of Rome
Lockless S.r.l.
marotta@diag.uniroma1.it
marotta@lockless.it

Davide Cingolani
Sapienza, University of Rome
Lockless S.r.l.
cingolani@diag.uniroma1.it
cingolani@lockless.it

Alessandro Pellegrini
Sapienza, University of Rome
Lockless S.r.l.
pellegrini@diag.uniroma1.it
pellegrini@lockless.it

Francesco Quaglia
University of Rome "Tor Vergata"
Lockless S.r.l.
francesco.quaglia@uniroma2.it
quaglia@lockless.it

ABSTRACT

The share-everything PDES (Parallel Discrete Event Simulation) paradigm is based on fully sharing the possibility to process any individual event across concurrent threads, rather than binding Logical Processes (LPs) and their events to threads. It allows concentrating, at any time, the computing power—the CPU-cores on board of a shared-memory machine—towards the unprocessed events that stand closest to the current commit horizon of the simulation run. This fruitfully biases the delivery of the computing power towards the hot portion of the model execution trajectory. In this article we present an innovative share-everything PDES system that provides (1) fully non-blocking coordination of the threads when accessing shared data structures and (2) fully speculative processing capabilities—Time Warp style processing—of the events. As we show via an experimental study, our proposal can cope with hard workloads where both classical Time Warp systems—based on LPs to threads binding—and previous share-everything proposals—not able to exploit fully speculative processing of the events—tend to fail in delivering adequate performance.

KEYWORDS

Discrete Event Simulation; PDES; Pending Event Set; Lock-free Synchronization; Speculative Simulation; Shared Memory; Share Everything

ACM Reference Format:
Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2018. The Ultimate Share-Everything PDES System. In *SIGSIM-PADS '18: SIGSIM Principles of Advanced Discrete Simulation CD-ROM, May 23–25, 2018, Rome, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3200921.3200931

1 INTRODUCTION

Parallel Discrete Event Simulation (PDES) [7] is a powerful methodology, which provides the support for simulating huge/large and complex discrete-event systems. PDES has been conceived in order to enable the exploitation of (massively) parallel computing systems. This is achieved by partitioning the simulation model into simulation objects (also known as Logical Processes - LPs) which are enabled to process simulation events concurrently.

Along its life, this methodology has been integrated with techniques and solutions aimed at continuously improving its capability to fruitfully exploit computing resources, with the ultimate objective of improving performance and scalability. However, until recently, the most of the literature techniques were based on recognizing an individual simulation object as the work unit within the optimization process. Consequently, optimizations of PDES have been essentially based on optimizing the run time dynamics of PDES systems under the common way of thinking that simulation events are not fully representative as individuals in the optimization process. Rather, aggregates of events have been seen as the *weight* to be assigned to a simulation object in order to determine how to manage it in the simulation run. As an example, an object targeted by sets of CPU-demanding events is typically considered as a heavy-weight object, a factor that has been considered as relevant to bind that object to a specific thread—or CPU-core—in order to enable balanced advancement of the logical time across all the objects (see, e.g., [3]).

In more recent times, the advent of multi-core shared-memory machines has generated a new way of devising optimizations of PDES. This is based on the idea that all threads—so all CPU-cores—running the PDES platform can fully share finer grain work units, namely individual events possibly bound to different simulation objects. This is the *share-everything* PDES paradigm conceived in [15]. It has the intrinsic advantage of enabling the delivery of the overall used computing power to the events that are—at any time—the closest ones to the current commit horizon of the simulation. In fact, this paradigm imposes no limitation on what simulation objects can be dispatched for execution by a thread along the simulation execution timeline—a limitation that is instead typical of common PDES systems based on partitioning the objects across worker threads. Such a new approach is simple in principle, in fact it is based on the concept of a fully-shared event pool containing the events destined to whichever simulation object, from which all the threads extract the higher priority events (those with timestamps closer to the current commit horizon) for processing, and into which the same threads put newly generated events.

On the other hand, fully sharing the workload of events across all threads poses hard problems in terms of managing the computing power in a truly scalable and effective way. In fact, a share-everything PDES engine capable of exploiting such computing power should guarantee that:

A) threads do not block each other while accessing shared data structures, namely the event pool and also the actual states of the simulation objects—otherwise thread synchronization would become a bottleneck;
B) threads do not wait for each other because of potential causality constraints between the simulation events (so the objects) they are currently running—otherwise virtual-time synchronization would become a major factor preventing hardware parallelism exploitation.

Overall, an ideal share-everything PDES platform should guarantee scalability along the following two dimensions in a combined manner: 1) wall-clock-time coordination across threads and 2) virtual-time coordination across simulation objects, ultimately managed by threads. While point 1) has recently been tackled by a few works [12, 16]—they provided non-blocking algorithms for managing the fully shared event pool and share-everything-suited event-dispatching rules that avoid collisions across threads in the access to the state of the simulation objects—a holistic design coping with both the above points in a combined manner is still lacking. Such a design is the objective of this article, where we present a share-everything PDES system that entails speculative execution capabilities of the simulation objects—guaranteeing scalability of virtual-time coordination—and fully non-blocking wall-clock-time coordination across threads.

Technically, our work provides solutions for a set of problems intrinsically related to the construction of speculative share-everything PDES systems, which were not tackled by the literature. They are:

- the definition of non-blocking algorithms for managing a fully-shared pending-event set that contains both schedule-committed events (those produced by the execution of other events that have been detected to be safe and causally consistent) and non-committed ones (those that are the result of speculative, not yet committed, processing actions), which might need to be (logically) canceled;
- the definition of non-blocking algorithms for dispatching the events to be processed across threads in such a way that threads never collide on a same simulation object and causal consistency is detected on-the-fly—also exploiting lookahead information—leading to the possibility to optimize the way events are actually processed (in terms of configuration of event-undo support). Consequently, our PDES system provides solutions for combining on a fine-grain basis (event by event), conservative and speculative processing techniques.

As a matter of fact, our share-everything PDES system can cope with hard-workload scenarios where there are (sudden) skews in the distribution of the events across simulation objects along virtual time. These skews possibly create relatively short bursts of events to be processed at a subset of the objects, while other objects have no (or few) events to be processed along that same virtual time window. In these scenarios traditional PDES-oriented load balancing approaches, based on medium-term binding between objects and threads, have scarce capability to react to the sudden unbalance that may materialize, which can lead to an increase of the likelihood of wasted computation in case of speculative processing. The share-everything paradigm that we adopt considers events as fully-shared workload units, thus being able to concentrate the computing power, say threads, on any burst of events that materializes among subsets of objects—any thread can in fact take care of processing whatever event in these bursts, thus contributing to promptly advance the currently hot portions of the simulation model. Furthermore, the speculative processing capabilities we include in our PDES system enable threads to process these bursts with no blocking phase along virtual time, as instead it occurs in previous share-everything proposals like [12].

On the downside, the price our share-everything PDES system pays stands in the impossibility to exploit large or extreme scale clusters of distributed memory resources, which can instead be exploited by traditional non-shared memory bound PDES engines [2]. However, the perspective of our design is strengthened by the always rising trend towards larger numbers CPU-cores on a same shared-memory chipset, motivated by the already reached power wall affecting the growth of the computing speed of individual CPU-cores. On the other hand, future PDES architectures could be envisaged where on each individual shared-memory machine an instance of our share-everything PDES platform could be run, and the instances could, in their turn, be clustered via additional coordination mechanisms on a distributed memory platform.

Our share-everything PDES system has been released as open source[1] and we also report experimental data for an assessment of our proposal in comparison with traditional PDES and previous share-everything solutions not entailing speculative capabilities.

The remainder of this article is structured as follows. In Section 2 related work is discussed. Section 3 presents the design of our *ultimate* share-everything PDES system. Experimental results are reported in Section 4.

2 RELATED WORK

Our proposal is along the path of building PDES systems that are optimized for execution on shared-memory machines. This topic has been addressed in the literature by several works and in compliance with various objectives. In [22, 24–26] the authors provide solutions for reorganizing traditional-style PDES systems, making them more suited for shared-memory platforms. Few solutions optimize the architecture of the communication facilities across the threads. Other solutions take advantage of the possibility for any thread to promptly access the state of any simulation object and of its event queue when a re-bind between objects and threads is needed—for load balancing—depending on the objects' current weight in the computation. In some case, interference from external workload is also considered in the re-bind. Our solution is completely different from these proposals since it is not based on traditional partitioning

[1]Source code available at `https://github.com/HPDCS/USE`
Artifact available at `https://doi.org/10.5281/zenodo.1196287`

of the workload (say the objects) across the threads, and on periodic re-evaluation of partitions. Rather, we consider any individual event as a work unit that can be dispatched along any thread in the PDES system.

The works in [5, 17, 18] exploit shared memory for enriching the PDES programming model in order to provide support for sharing information across simulation objects. This objective is achieved by means of transactional memory, software instrumentation or operating system facilities. However, these proposals are still bound to the traditional PDES paradigm. In fact, they have been integrated into environments that still rely on object (namely workload) partitioning across threads, rather than fine-grain sharing of individual work units—single events—like in our approach.

Clearly, the solution we provide also stands along the path of building environments where threads coordinate in the access to the fully-shared event pool—or more generally to shared data structures—in a highly scalable manner. The topic of providing event-pool data structures enabling concurrent accesses has been addressed in [1], which proposes an approach based on fine-grain locking of a sub-portion of the data structure upon performing an operation. However, the intrinsic scalability limitations of locking still lead this proposal to be not suited for large levels of parallelism, as also shown in [19]. Rather, in our ultimate share-everything PDES system we base concurrent accesses to shared data structures on non-blocking algorithms, which have been shown to be much more prone to scalability.

As for non-blocking management of sets by concurrent threads, various proposals exist, such as lock-free linked lists [10], skip-lists [20] and Calendar Queues [15, 16]. A few of these proposals, like [15, 16], have been exploited as building blocks in the share-everything PDES paradigm. However, the outcoming solutions do not fit scenarios where two or more threads pick from the shared-event pool events destined to a same simulation object. In these scenarios, threads still block each other because of a critical section implementing the processing stage of the events at the destination object, which limits scalability especially with workloads entailing event bursts at subsets of objects. This problem has been tackled in [12], where non-blocking event-pool management is combined with CPU-dispatching rules that avoid collisions of multiple threads on the state of a same simulation object. However, differently from our proposal, none of these solutions guarantees non-blocking coordination in virtual time. In fact, they are all based on a kind of *wait-until-validated* paradigm, which does not allow multiple events to be processed speculatively on a same object as in Time-Warp style [13]—in fact, just one event can be processed speculatively at each object, and is then committed (namely validated) or rolled back depending on blocking virtual-time synchronization conditions, before any other event at that same object can be CPU-dispatched. In our solution, we enable Time-Warp style speculation, while still keeping all the advantages from non-blocking thread coordination in the access to shared data structures along wall-clock time.

Non-blocking operations on event pools have also been studied in [9], which presents a variation of the Ladder Queue where the elements are at any time bound to the correct bucket, which is an unordered list. The extraction from an unordered bucket returns the first available element, which does not necessarily correspond to the one with the minimum timestamp. This proposal is intrinsically tailored for PDES systems relying on speculative processing, where unordered extractions leading to causal inconsistencies within the simulation model trajectory are reversed (in terms of their effects on the simulation model trajectory) via rollback mechanisms. In our proposal we guarantee the ordering of the events in the shared pool, which allows us to put in place the smart combination of conservative (say safe) and speculative processing at the level of each individual event—also thanks to the explicit exploitation of the lookahead in the simulation model—thus enabling the optimization of the rollback support, an aspect that is not considered in [9]. Also, in this work the non-blocking data structure is essentially used as a CPU-dispatching support allowing threads to pick the next event of some object concurrently with other threads. However, differently from our present proposal, binding mechanisms between sets of objects and sets of threads are still considered, thus making the approach not fully compliant with the share-everything paradigm.

The recent proposal in [11] explores the idea of managing concurrent accesses to a shared pool by relying on Hardware Transactional Memory (HTM) support. Insertions and extractions are performed as HTM-based transactions, hence in non-blocking mode. However, the level of scalability of this approach is limited by the level of parallelism in the underlying HTM-equipped machine, which nowadays is relatively small. Also, HTM-based transactions can abort for several reasons, not necessarily related to conflicting concurrent accesses to a same portion of the data structure. As an example, they can abort because of conflicting accesses to the same cache line by multiple CPU-cores, which might be adverse to PDES models with, e.g., very large event pools. Our proposal does not require special hardware support, thus fully eliminating the secondary effects caused by, e.g., HTM limitations on the abort rate of the operations.

3 THE PDES SYSTEM

As in classical PDES, our system supports models that are partitioned in simulation objects whose execution is carried out by Logical Processes (LPs). LPs are sequential entities and a specific LP is CPU-dispatched when a Worker Thread (WT), triggers the execution of the handler of an event destined to it. A share-everything arrangement of the PDES platform leads to a scenario where: (i) an LP can be CPU-dispatched by whichever WT at any point of the simulation execution; (ii) all the events are maintained by a unique pool fully shared across all the WTs.

In our design, the system has two main data structures:

(A) a set of *LP Control Blocks* (LPCBs), which are used to keep metadata representing the system-level view of the advancement of the LP in simulation time—this is a concept disjoint from the actual application level state of the simulation object encapsulated by the LP;

(B) a set of events—either already processed, or to be processed, or logically canceled—maintained into the aforementioned fully-shared unique pool which we refer to as *Scheduling Queue* (SQ).

At first approximation, the main execution loop carried out by all the WTs consists in: i) fetching some event to be processed from the SQ; ii) performing a rollback of the target LP, if required;

iii) executing the event; iv) updating the LPCB; v) inserting newly generated events into the SQ.

As for point i), the fetch operation is contextual to the try-lock of the target LP. Hence, no WT will ever fetch an event bound to an LP that is currently locked by some other WT. Overall, no mutual block among WTs will ever occur in the attempt to access the same LP, since the WT that will experience a failure of its try-lock operation will simply go ahead scanning the event pool in order to take an event destined to some other LP. This also guarantees isolation of WTs' accesses to a given LPCB and to the corresponding simulation object state, in both forward and rollback mode.

Concerning the other operations accessing the SQ within the main loop, as we will discuss, they are all carried out in a non-blocking fashion—including secondary updates on individual nodes' data in order to correctly represent their state (e.g. logically cancelled because of a rollback) within the speculative processing scheme, as discussed in the following.

3.1 Architectural Details

3.1.1 LP Control Blocks. Each LPCB is formed by a set of variables which hold metadata needed by the simulation engine to detect any relevant runtime condition related to the LP, including its involvement in causality errors. Noteworthy, among others, the LPCB keeps a pointer named `bound` to the last processed event, whose timestamp represents therefore the Local Virtual Time (LVT) of the simulation object associated with the LP. The actual buffer keeping the event pointed by `bound` still stands in the SQ, so that event processing leads to no actual removal from that queue. Unlinkage from this queue will occur when we detect that the event is either definitively causally consistent—it is a committed event—or it should not appear along the execution—it is an event generated by a rolled back one. Discriminating whether the event has been processed, or it is in a different state with respect to the state of the target LP and of all the other LPs, will take place via a proper state machine coded into the event-buffer metadata. The state diagram for this state machine will be discussed shortly.

The LPCB also maintains a non-negative integer named `epoch` which keeps track of the total number of rollback operations the LP has experienced. This information essentially tells in what incarnation the LP is currently executing along its forward path, given that a rollback leads to a new incarnation—a new LP life after the causality error.

Moreover, each LPCB keeps metadata to retrieve what we call the `local_queue` of the simulation object, which is used to maintain the history of processed events at that LP. Those that are rolled back are not included in this queue. Also, `local_queue` is built in our system as a view of event-buffers associated with the LP which are anyhow kept within the SQ. In other words, `local_queue` is built by relying on cross event-buffers' linkage standing aside of their linkage into the SQ. The reason for having such a view, rather than only relying on the global view of events in the SQ, stands in the management of both state reconstruction—which in our system is based on checkpointing and coasting forward—and CPU-dispatching of the next-to-be-processed event of the LP. Finally, the LPCB keeps the lock actually used to support try-lock operations when WTs try to take on the job of working on the LP.

3.1.2 Events Representation. In our PDES system, an event is a simple memory buffer—the event-buffer—exchanged between two LPs, or sent from some LP to itself. The actual exchange takes place through insertion and extraction operations to/from the SQ. Each event originates on one LP, called *sender*, and targets another LP, called *receiver*, which could be the same event's source LP.

Each event-buffer is made up by (a) metadata—used by the PDES system for treating the event—and by (b) the actual payload conveying the information to be delivered to the event-handler for application level processing. In this section we focus our attention on metadata, given that our system is application agnostic, and can support generic simulation models.

An event-buffer associated with the event e keeps the following metadata:

- the id of the LP which generates and sends the event, and the id of the LP which must receive and execute it, respectively hold by `sender` and `receiver` fields;
- the timestamp `ts` at which the event must occur along simulation time;
- a field `epoch`, which maintains the epoch of the receiver LP at the time when it processed the event;
- a pointer `parent` to the event p, whose execution has generated e;
- a field `parentEpoch`, which represents sender LP's epoch when p has been processed, namely the incarnation number of the sender LP at the generation time of e;
- a variable `state` used to represent the current state of the event within a finite-state machine, which drives the event management logic at the level of the PDES system.

Since our system entails speculative execution capabilities, possible violations of timestamp order might occur, and rollback operations are required in order to restore the correct execution trajectory of an LP (a timeline), as well as to undo the production of new events along the incorrect trajectory. In general, an event experiences several life stages. We define as *committed* the simulation trajectory of an LP that is observable at the end of a concurrent execution entailing no timestamp order violation. Every event that is visible within that simulation trajectory, is defined as *committed* or *safe*. On the contrary, events could be *retracted*, meaning that they cannot longer exist in any time-line.

In our system an event being processed flushes newly produced events to the SQ before the execution phase is over. Therefore, the unprocessed (or being-processed) event with the minimum timestamp is a safe event that corresponds to the commit horizon, namely the Global Virtual Time (GVT) of the speculative run.

Given that, thanks to the try-lock mechanism, two events destined to the same LP cannot be concurrently processed by WTs, we can exploit the lookahead (LA) of the simulation model to compute safety of whatever event to be processed (or being processed) according to the following expression:

$$\begin{aligned} is\_safe(e) \quad = \quad & (e.\texttt{ts} \in [GVT, GVT + LA) \ \wedge \\ & \nexists\, e' : \ e.\texttt{lp} = e'.\texttt{lp} \ \wedge \ e'.\texttt{ts} < e.\texttt{ts}) \end{aligned} \tag{1}$$

If an event e is not safe—meaning that Equation 1 does not currently hold for it—and gets speculatively processed, its execution might be undone because of the arrival of a straggler destined to the same

LP. However, in such a scenario, the event e could be still *valid*, meaning that it is requested to appear as executed along some timeline of the destination LP. On the other hand, if the event was generated by some other event that is undone, the former becomes *invalid*. In fact, it should no way appear in the correct timeline of the destination LP—in classical Time Warp these are events canceled by their corresponding anti-events.

Such as for the safety, our system detects if a particular event e has become invalid at a given point in wall-clock time by exploiting event-buffer metadata. In particular, an event e is currently valid if and only if (i) its parent p is currently valid as well and (ii) the execution of p that generated e has not been undone. Exploiting event metadata, the definition of validity can be formalized by the following recursive function:

$$is\_valid(e) = \begin{cases} true, & \text{if } e.\mathtt{type} = \mathtt{INIT} \\ (e.\mathtt{parentEpoch} = e.\mathtt{parent.epoch}) \wedge & \\ is\_valid(e.\mathtt{parent})], & \text{otherwise} \end{cases} \quad (2)$$

The above formalization is based on having the validity of an event always depending on the validity of its parent. The unique exception is the INIT event —used to just setup the simulation initial state, including the states of the LPs— which is safe (hence valid) by construction.

To check whether an event e has been re-executed or not, we harness the parentEpoch information kept by the event-buffer, which is compared to the epoch of its parent p. As said before, e.parentEpoch and p.epoch have been set with the epoch of the sender LP at the time the event p has been executed. Since the LPs' epochs are updated after a rollback takes place, if an event is re-processed, its epoch number will be updated with the new LP's epoch. As soon p.epoch is not equal to—more precisely greater than—e.epoch, it means that the parent is living within a new and different timeline, to which the child event e does no longer belong. Consequently, if e.parentEpoch = p.epoch we can infer that the event p has not been re-executed, and therefore still stands on the original timeline that generated e.

In our share-everything PDES system we do not immediately unlink events from the SQ when they become invalid. This is because we manage the queue via non-blocking algorithms. As a consequence, a node in the queue—namely, an event-buffer—might be required to still stand into the queue to facilitate the execution of non-blocking queue traversals by WTs. In fact, they can use that node as a link between others, even though the corresponding event appears to be as no longer relevant for the execution of any LPs' timeline. Also, temporarily keeping invalid event-buffers into the SQ is a way to asynchronously notify the other WTs currently traversing the SQ that something has changed along the timeline of some LP—since invalid event-buffers expose updated metadata—which may in its turn drive the actions by these same WTs. In other words, each single node in the SQ is associated with a state machine that helps supporting a fine grain coordination across WTs, implemented according to the non-blocking paradigm.

Figure 1 shows the actual state machine within which each event-buffer lives. How state transitions occur based in the pseudo-code executed by WTs will be discussed in Section 3.2, while in this section, we illustrate the "meaning" of each state. A newly produced event, just inserted into the SQ, is born with a clean state (CLN) representing that no operation has been performed yet on the event-buffer—except its insertion, clearly with the correct metadata such as the epoch of the sender LP. Note however that, starting from the wall-clock-time instant the event appears to be incorporated into the queue, any WT can be already traversing it, possibly updating its state. When a WT finds an event-buffer e marked with the CLN state in the SQ, it logically extracts (fetches) e, by marking it as FTCH—we recall that for this to occur, the WT must have observed the presence of the event-buffer, and must have successfully executed the try-lock on the target LP. In fact, two different WTs cannot take on the job of concurrently processing a same event destined to a given LP, which would require special features in the programming model (and in its runtime) that are aside of the work we are presenting. Marking an event-buffer as FTCH leads to notify that the event is currently in charge to some WT.

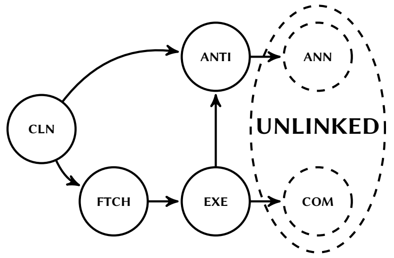


Figure 1: State diagram for the event's life-cycle.

It is possible that, when the WT successfully try-locks the LP associated with an event-buffer in the CLN state, this event has already become invalid, since the execution of its parent might have been undone or invalidated. When a WT observes this condition while traversing the SQ, it makes the event transit to the logical ANTI state, whose name evokes the anti-event concept—meaning that it simply does no longer belong to the simulation trajectory, along any timeline. All the state transitions are implemented atomically in our software, given that they can be carried out by concurrent WTs. For potentially conflicting (mutually excluding) transitions—such as CLN⟶ FTCH and CLN⟶ ANTI—we rely on the Compare-and-Swap (CAS) machine instruction. For non-conflicting ones, we simply use atomic memory writes—such as test-and-set—abstracted by the SET statement in the pseudo-code. Therefore, the transition to ANTI excludes the possibility for a WT to successfully transit the node to FTCH. It is clear that an event still complying with the validity rules expressed by Equation 2, might really be no longer valid. It is only a matter of time for the WTs to detect that such information related to validity is reflected into the state of the parent event. On the other hand, speculative processing is already known to accept the risk of processing something that seems to be consistent, in terms of timestamp ordering, but which is actually no longer consistent given that something is happening concurrently along model execution.

An event successfully marked as FTCH is returned to the main loop of the WT, where it will be processed, as we shall describe. Here, the event transits to the "execution" state, say EXC, meaning that the event-handler actually took it for performing the corresponding LP state manipulations.

Overall, the ANTI state, still reachable from the EXC state, is used to discriminate that the PDES system knows that the event does no longer belong to any valid LP timeline, either if it has been already fetched and executed by some thread—thus it passed through EXC—or if it is found to be invalid prior being fetched. The ANTI state is reached via a transition from EXC when a rollback occurs related to the passage of the event to the invalid state. Therefore, it will not need to be re-processed after the rollback. The PDES system can detect that a rollback needs to be executed when a WT traverses the SQ comparing the metadata of the LP and those kept by the event-buffer (such as the current LVT of the LP and the event timestamp, or its parent's state). These checks are anyhow executed without the need for locking the target LP. If the event marked as ANTI is found to stand in the future—or on a new timeline after a rollback of the target LP—the event is simply transited to the ANN state, an absorbing state leading to the unlink of the event from the SQ.

Similarly, when an event ends its life-cycle and appears along the correct LP timeline, it is logically marked as COM (commit state). Clearly, an event transits to COM after being processed if it is found to be a safe one. In this case it is also unlinked from the SQ. Although unlinked from the SQ, an event-buffer in the COM state will be garbage collected (reused) successively, as we shall discuss, since some child could still refer to it for validity assessment according to Equation 2. Another motivation for retaining the event is its usefulness for state reconstruction purposes in a rollback phase, as we will also discuss.

As a final note, an event can persist in the EXC state across multiple executions, caused by rollbacks, up to the point in time when its safety is assessed, or it becomes invalid.

3.1.3 Scheduling Queue. The Scheduling Queue (SQ) used in our share-everything PDES system is a conflict-resilient lock-free priority queue that sorts event-buffers (across all the LPs) on the basis of their timestamps. In particular, it is a Calendar Queue supporting non-blocking operations. We borrow its implementation from [16], reshuffling it in order to meet the needs of our innovative share-everything PDES system. At a logical level, such a queue can be abstracted as a generic non-blocking ordered linked list like the one proposed by Harris [10]—although being much more efficient thanks to its multi-bucket organization leading to amortized constant-time access. Relying on this abstraction allows us to hide the complexity of non-blocking Calendar Queue operations, which are not the focus of this article—jointly enabling us to focus on how we exploit non-blocking capabilities of such priority queue in our PDES system.

In our reshuffle, the priority queue has an ENQUEUE API and two other primitives: GETMIN, which retrieves a pointer to the event with the smallest timestamp which is still linked to the queue, and GETNEXT, which retrieves the pointer to the event which immediately follows—along virtual time—the one identified by its input argument. Thanks to this support we can build a cross-layer optimized FETCH operation that returns in a non-blocking mode a to-be-processed event associated with some LP not currently locked by any WT (see Section 3.2.2 for the details). Further, the SQ supports an UNLINK API which is used to disconnect a generic event from the SQ—those that transit to the ANN or COM state—still in non-blocking fashion.

Before returning an event e, a WT executing a FETCH executes a try-lock operation on the target LP. If this operation fails, the WT slides to the subsequent event in the queue—the one successive to e. This is done thanks to the exploitation of the above mentioned get services in the queue API. This sliding scheme is iterated up to the point where a try-lock on the LP targeted by some event, encountered along the queue, executes successfully.

According to the event's state diagram in Figure 1, fetched (or already processed) events are not unlinked from the SQ. In fact, an event could be re-executed due to a rollback. Hence, it must be visible in future queue explorations by concurrent WTs in order to be properly handled. Moreover, our event validity definition (see Equation 2) implies that parent metadata could be accessed while assessing the state of its children. Therefore, the actual garbage collection of the event-buffer—leading to its reuse—is also determined by the relation between the GVT value and the timestamp of child events, as we shall discuss. On the other hand, the logical removal of the node in the COM state associated with the minimum timestamp value from the SQ via the UNLINK API is enough to move forward—beyond that node—the pointer to the new minimum timestamp element into the queue. As said, this is because in our PDES system the removal of that COM event, which was previously processed, already led to incorporate into the SQ all its children, if any.

3.2 Worker-Thread Algorithm

3.2.1 Main loop. The pseudocode of the main loop carried out by any WT is shown in Algorithm 1. Initially a call to the FETCH procedure is executed to retrieve from the SQ an event to be handled (processed or undone/retracted), which is destined to an LP not currently in charge of another WT—namely, locked by the caller WT. The FETCH procedure returns to the caller WT a pointer to the event to be handled, and the indication of whether the event is safe (computed according to Equation 1) or at least valid. The FETCH procedure also returns the minimum timestamp of the non-committed event standing into the SQ, namely the current GVT value. Further, according to state transitions, such procedure may lead events destined to whatever LP to transit from CLN to ANTI while traversing the SQ. This is based on validity checks as expressed by Equation 2.

Regardless the retrieved event's current state, the thread checks if its timestamp is smaller than the LP's LVT, namely the time reached by the LP executing the event pointed by its bound variable in the LPCB. In the positive case a ROLLBACK is triggered in order to bring back the LP's state to the timestamp of the last event preceding the retrieved straggler event and belonging to the current timeline. At the end of the ROLLBACK procedure the simulation state is compliant with respect to the execution of the currently retrieved event. If this event was already marked as EXC, it simply persists into this state. Otherwise, it will transit from FTCH to EXC.

Before the execution of the event, a MAKEUNDOABLE operation is called, which implements whatever policy for making the current state transition undoable (namely, rollbackable). In our implementation we opted for a traditional periodic checkpointing approach. Based on the selected policy, if the log is taken, the log-node is linked to the corresponding event (the one pointed to by bound), for correct alignment of the data structures. We note that the safety information associated with the event retrieved via FETCH can be

Algorithm 1 Simulation Loop

```
1:  procedure SimulationLoop()
2:      $<evt, safe, valid, gvt> \leftarrow$ Fetch()
3:      $curLP \leftarrow evt$.receiver
4:      if $evt \neq$ null then
5:          if $evt$.ts < $curLP$.bound.ts then
6:              Rollback($curLP$, $evt$.ts)
7:          if $\neg valid$ then
8:              SET($evt$.state$\leftarrow$ ANTI)
9:          else
10:             MakeUndoable($curLP$,$evt$,$safe$)
11:             LinkToLPQueue($curLP$.bound, $evt$)
12:             $evt$.epoch $\leftarrow$ $curLP$.epoch
13:             $newEvts \leftarrow$ Execute($evt$)
14:             Flush()
15:             $curLP$.bound $\leftarrow$ $evt$
16:             SET($evt$.state, EXC)
17:         Unlock($curLP$.lock)
18:         if $safe$ = TRUE then
19:             Unlink($evt$)
20:     GVTOperations($gvt$)
```

exploited for optimizing the management of the undo support. In our case, it could lead to take a checkpoint and to rejuvenate the checkpoint period if the event will never be undone—this will enable reducing coasting forward overhead by inserting a log exactly on the currently committed event of the LP. On the other hand, if reverse computing approaches were employed (see [4, 6]), the safety information would simply tell that there will be no need to generate data/metadata for reversing the event execution. The relation among the different data structures we used to manage each individual LP is schematized in Figure 2. By the scheme we see how the event queue of the LP—namely `local_queue`—is incorporated into the SQ—which is global to all the LPs. Thus, the LP local queue represents a sort of view on such global queue achieved via a parallel linked list. The processed event is linked to the per-LP view of the SQ via LinkToLPQueue, so as to make it immediately available for any purpose, including coasting forward if requested.

The real event-processing phase starts by updating the event epoch with the LP current one, in order to represent its current incarnation within the LP current timeline. Then, the event is executed by invoking the application-level event-handler provided by the simulation model. New events possibly produced along the event-handler execution are stored in a local buffer, in order to Flush them to the SQ at the end of the current event execution. At this point, to complete event processing, the event state is atomically set to EXC and the LP bound is updated in order to point to the current event.

The order of those operations is fundamental to guarantee correctness. In fact, newly produced events are flushed into the SQ before their parent's processing is finalized, otherwise we would violate the definition of GVT we rely on. So, it is important to postpone the update of the current bound and of the state of the event after the flush operation.

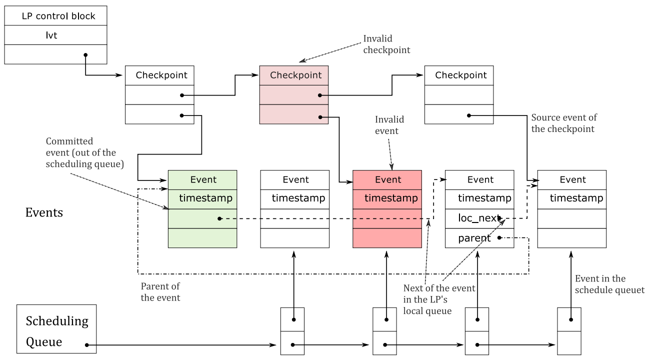


Figure 2: Data structures of LP state, SQ and events, and relative relationships.

Clearly, once updated the event state, the WT releases the lock on the target LP, thus enabling concurrent WTs to eventually take care of whatever event destined to the same LP, as in the spirit of the share-everything paradigm. Finally, if the Fetch procedure indicates that the event is a safe one, the final logical transition to the COM state is directly executed by the in-charge WT, which unlinks the event from the SQ.

At the end of the main loop, despite what happened before, some housekeeping tasks are performed, which take place via the GVTOperations procedure. Specifically, this procedure is used to reclaim memory associated with no longer useful event-buffers, as well as checkpoints. As we will clarify while explaining the structure of the Fetch procedure, event-buffers are initially unlinked from the SQ when they will no longer need to be handled, which means they are committed or have been annihilated, transiting respectively in COM and ANN states. On the other hand, the event-buffers in the COM state still remain into the per-LP `local_queue`, while the event-buffers in the ANN state are inserted (upon their unlink from the SQ) into per-WT retirement queues. All these event-buffers can be actually reused (hence reclaimed by the memory system) only after the GVT value oversteps the maximum timestamp of any child possibly generated by their execution. This is because validity of an event-buffer in our system is based on referring the state of a parent event-buffer according to Equation 2. To easily keep track of such condition at runtime, each event-buffer is also filled with a timestamp field which exactly keeps such a maximum child-timestamp. While executing the GVTOperations procedure, the WT deallocates all the event-buffers from its list of ANN nodes which satisfy such condition. The same occurs for the nodes in the `local_queue` of the LP that the WT is currently in charge of.

3.2.2 *The Fetch Procedure.*

The Fetch procedure, whose pseudocode is shown in Algorithm 2, has two objectives: (i) returning an event to be processed by the WT main loop; (ii) unlinking no-longer needed events from the SQ and hence from the per-LP views of this queue. This procedure uses two local variables *gvt* and *jmpLP*. The former keeps the timestamp of the minimum event still linked to the SQ, the latter keeps the set of LPs targeted by "skipped" events—the events that have been traversed by the WT without actually locking the target LP.

Algorithm 2 Fetch procedure

```
F1:  procedure FETCH()
F2:      $evt \leftarrow$ GETMIN()
F3:      $gvt \leftarrow evt$.ts
F4:      $jmpLPs \leftarrow \{\}$
F5:      while $evt \neq$ null do
F6:          $LP \leftarrow evt$.receiver
F7:          $evtState \leftarrow evt$.state
F8:          $safe \leftarrow is\_safe(evt, gvt, jmpLP)$
F9:          $in\_past \leftarrow evt$.ts $\leq LP$.bound.ts
F10:         $valid \leftarrow is\_valid(evt)$
F11:         if TRYCLEANANDSKIP($evt$,$jmpLPs$,
F12:                     $LP$,$safe$,$in\_past$, $valid$) then
F13:             if TRYLOCK($LP$.lock) then
F14:                 $curr \leftarrow$ GETLOCALNEXTANDVALID($evt$)
F15:                 if $curr \neq evt$ then
F16:                     $valid \leftarrow true$
F17:                     $safe \leftarrow is\_safe(curr, gvt, jmpLP)$
F18:                 $in\_past \leftarrow evt$.ts $\leq LP$.bound.ts
F19:                 if $\neg valid$ then
F20:                     if $in\_past$ then
F21:                         return $\langle evt, safe, gvt \rangle$
F22:                     SET($evt$.state$\leftarrow$ ANTI)
F23:                 $evtState \leftarrow$ CAS($evt$.state, CLN, FTCH)
F24:                 switch($evtState$)
F25:                     case CLN:
F26:                         return $\langle evt, safe, valid, gvt \rangle$
F27:                     case EXC:
F28:                         if $\neg in\_past$
F29:                             return $\langle evt, safe, valid, gvt \rangle$
F30:                         else if $\neg safe$
F31:                             $jmpLPs \leftarrow jmpLPs \cup \{LP\}$
F32:                         break
F33:                 RELEASELOCK($LP$.lock)
F34:             else
F35:                 $jmpLP \leftarrow jmpLP \cup \{LP\}$
F36:         $evt \leftarrow$ GETNEXT($evt$)
```

Algorithm 3 TryCleanAndSkip procedure

```
C1:  procedure TRYCLEANANDSKIP($evt$,$jmpLPs$,$LP$,
C2:                              $safe$,$in\_past$, $valid$)
C3:      $tryLock \leftarrow true$
C4:      if $valid$ then
C5:          if $in\_past$ then
C6:              if $safe \wedge evtState$ = EXC then
C7:                  UNLINK($evt$)
C8:              else
C9:                  $jmpLPs \leftarrow jmpLPs \cup \{LP\}$
C10:             $tryLock \leftarrow false$
C11:     else
C12:         if $evtState$ = CLN then
C13:             $evtState \leftarrow$ CAS($evt$.state, CLN, ANTI)
C14:         if $evtState$ = ANTI then
C15:             UNLINK($evt$)
C16:             $tryLock \leftarrow false$
C17:     return $tryLock$
```

Algorithm 4 GetLocalNextAndValid procedure

```
G1:  procedure GETLOCALNEXTANDVALID($curr$)
G2:      $LP \leftarrow curr$.receiver
G3:      $lNext \leftarrow$ GETLOCALNEXT($LP$.bound)
G4:      while $lNext \neq$ null $\wedge \neg is\_valid(lNext)$ do
G5:          SET($evt$.state$\leftarrow$ ANTI)
G6:          UNLINK($evt$)
G7:          $lNext \leftarrow$ GETLOCALNEXT($LP$.bound)
G8:      if $lNext \neq$ null $\wedge$ $lNext$.ts $<$ $evt$.ts then
G9:          return $lNext$
G10:     return $curr$
```

First, the FETCH procedure retrieves the current minimum from the SQ by invoking GETMIN and storing its timestamp into *gvt*. Also, it initializes *jmpLP* as an empty set. Then, for each traversed event *evt* a safety check is performed—so we check if $evt.\texttt{ts} \in [gvt, gvt + LA)$ and $LP \notin jmpLP$, where *LP* is the receiver of *evt* (this is the implementation of the check on the safety condition expresses in Equation 1). Moreover, we check if *evt* is in the past of the *LP* timeline by comparing the timestamps associated with *evt* and the *LP* bound. Then, TRYCLEANANDSKIP is executed. This is a non-blocking procedure aimed at unlinking from the SQ events that have expired their lifetime (they are into an absorbing state, namely COM or ANTI) or telling whether the current event has to be processed by returning a boolean value set to true. The latter case is associated with any event, which is not in absorbing states, and requires the target *LP* to execute it and/or to rollback the *LP* state. All these checks are carried out by the TRYCLEANANDSKIP procedure relying on the metadata we included in our event-buffers and in the LPCBs. If TRYCLEANANDSKIP returns true, then the WT try-locks the target LP. If this fails, then the WT skips to the subsequent event into the SQ, by relying on the GETNEXT API. This pattern is iterated up to the point where the WT successfully locks a target LP, or the tail of the SQ is reached—GETNEXT returns `null`. While traversing the SQ, the *jmpLP* variable is populated, keeping track of all the LPs for which the WT observed something to be present into the SQ, until some target LP is locked. This set is used to compute the safety of an event.

When some target LP is successfully locked, *evt* is checked again to determine whether it is in the past of the LP. This is because in the wall-clock-time interval between the first check on *evt* performed by TRYCLEANANDSKIP and the current processing phase, some other WT may have changed the actual bound of the target LP.

Then WT invokes GETLOCALNEXTANDVALID (Algorithm 4), that returns an event *lNext* which is either the first valid event following the bound of *LP* into its local view of the SQ—the `local_queue`—or the event just fetched from the SQ. We need this procedure to check whether some event that is subsequent to the bound into `local_queue` has a timestamp lower than the one just fetched from the SQ, which represents a critical scenario to cope with. Such a scenario is illustrated via an example shown in Figure 3. Suppose that WT *A* holds a lock on an LP *X* because it is processing an event *e*. If another thread *B* tries to acquire the lock on *X* for processing an event *f* such that *e* precedes *f* ($e \rightarrow f$), it will fail and continue to analyze the next event *g*. This event is such that $e \rightarrow f \rightarrow g$ and

Wall-clock time	Thread A	Thread B
1:	`Check Event A on LP 1`	
2:	`Trylock on LP 1`	
3:	`Success`	`Check Event A on LP 1`
4:	`Process Event A`	`Trylock on LP 1`
5:	.	`Fail`
6:	.	`Check Event B on LP 1`
7:	.	`Trylock on LP 1`
8:	.	`Fail`
9:	`Release lock on LP 1`	`Check Event C on LP 1`
10:		`Trylock on LP 1`
11:		`Success`
12:		`Process Event C`

Figure 3: The scenario tackled by GetLocalNextAndValid.

targets X, so B will try to acquire again the lock on that LP. If in the meanwhile A has released the lock on X, B takes the lock and starts processing g. Supposing that e, f and g are all valid events, B executes g moving forward the bound of LP X. If f has a CLN state, executing g is not problematic since some WT eventually retrieves f and executes it after triggering a rollback as if it were a straggler event. Conversely, if f has an EXC state, no WT will ever execute such event again since it appears to be in the past of the current trajectory. This situation is prevented to occur exactly by the presence of GetLocalNextAndValid, that searches for an event (f) with higher priority than the currently fetched one (g) from the SQ.

Then, if the GetLocalNextAndValid procedure returns a different element with respect to the one originally identified, the relative validity information is updated. Thus, the first performed check is about the event validity—note that this check is carried out outside the locking region of the target LP. Since in the TryCleanAndSkip procedure invalid and CLN events are correctly marked (as ANTI) and unlinked, here we can assume that if we met an invalid event-buffer, it is not a newly inserted one. In this case we have to perform a rollback if and only if it is in the past of the current incarnation (epoch) of *LP*, otherwise we atomically set its state to ANTI, in order to notify that the event no more needs to be processed.

Otherwise, if the event was observed to be valid, WT tries to atomically apply the state transition CLN⟶FTCH with the CAS instruction. Of course, if it has been concurrently marked as ANTI by another WT that has seen the event as invalid, the CAS will fail.

After reading the actual state value of the event as a result of the CAS instruction[2], a switch case on it is carried out, implementing the events' finite-state machine discussed in Section 3.1.2. If the state is CLN, we know that the event has never been fetched and executed, thus, it is directly returned to the main loop. As discussed, it is up to the main loop to check if it is a straggler or not and trigger a rollback if required. The second case is the one where we have an EXC state, meaning that the event has been executed at least once. It follows that, it can be re-executed if and only if it is beyond the *LP* bound, meaning that it is in the future of the actual incarnation of the *LP* trajectory, namely the *LP* timeline. This is an event that has been rolled back and is still valid in the current (refreshed after the rollback) timeline. If the event is committable, namely if it is safe, we can unlink it from the global queue, otherwise we can skip the event by adding the *LP* to the *jmpLP* set, releasing the lock and retrieving another event.

Once the switch case has been executed, we can release the lock on the target LP, since here the event state can only be set to ANTI or EXC. In any case, the event unlink, that transits the event to the COM or ANN state, will be performed by any WT in TryCleanAndSkip, thus completing the event life-cycle.

[2]As in most common implementations, we assume that CAS returns the original value of the targeted memory location independently of whether its update fails.

4 EXPERIMENTAL ASSESSMENT

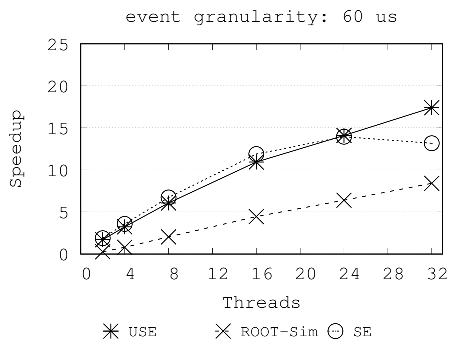
In this section we present performance results for a comparison of our Ultimate Share-Everything PDES system—which we refer to as USE—and the last generation Share-Everything solution presented in [12]—which we refer to as SE. The latter does not entail Time-Warp style processing of the events since, for each LP, at most one event is executed speculatively, and is eventually committed or aborted before any other event for the same LP can be CPU-dispatched. Its unique event pool—fully shared across WTs—only keeps so called *schedule-committed* events, which all need to appear along the LP timeline, thus not requiring the non-blocking management of any state machine for determining their actual role (across multiple ones) along model execution (e.g. if they need to be retracted because of the rollback of the parent). In other words, SE is blocking in virtual-time synchronization, while USE is fully non-blocking in both wall-clock-time thread coordination, and virtual-time LP synchronization, also thanks to its more sophisticated—and still non-blocking—logic for the management of the fully-shared event-pool data structure. For completeness of the analysis, we also include another competitor, which is the ROOT-Sim last generation traditional-PDES environment [23] not based on the share-everything paradigm—it adopts partitioning of the LPs across threads, with dynamic rebind for load-balancing purposes. It anyhow entails optimizations in its internal organization suited for shared-memory machines [24].

All the tests have been run on a 32-core HP ProLiant machine equipped with four 2GHz AMD Opteron 6128 processors and 64 GB of RAM. Each processor has 8 physical cores that share a 12MB L3 cache (6 MB per each 4-cores set), and each CPU-core has a 512KB private L2 cache. The machine is equipped with 64 GB of RAM—organized in 8 NUMA nodes—and we used Linux (kernel 3.2) as operating system.

The number of WTs running within all the used PDES systems we are comparing has been varied from 1 to 32 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.

4.1 Results with PHOLD

As first test-bed application we used the classical PHOLD benchmark [8] 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 set the loop to give rise to events with granularity of the order of 60 or 120 microseconds, thus spanning between fine-to-mid values leading to a representative setting for testing



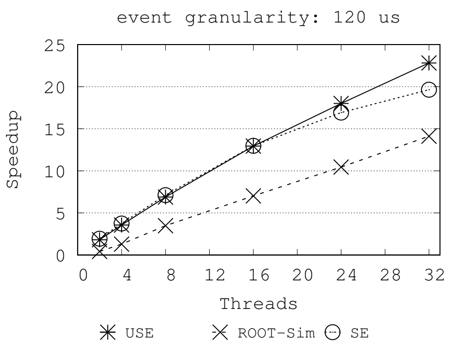
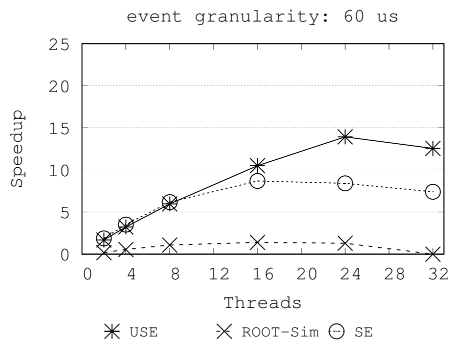


Figure 4: PHOLD speeup results - no hot-spot.



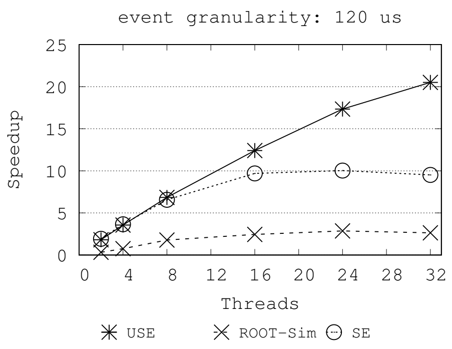


Figure 5: PHOLD speedup results - hot-spot.

parallel processing platforms—larger grain events might mask platform level costs for parallelization/coordination/rollback-support independently of the used PDES paradigm, share-everything vs traditional.

In one PHOLD configuration we included 10 hot-spot LPs, towards which 50% of the events injected by the other LPs are routed. 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—load-balancing, as in ROOT-Sim, can anyhow mitigate this problem. Also, they are difficult to manage in share-everything PDES systems where WTs can block one another because of the need to process events at a same LP (the hot-spot one) and the need for waiting the advancement of the spots before being able to advance the other LPs because of blocking (non-Time Warp style) virtual-time synchronization, as it may happen in SE. We decided to experiment with this kind of complex workload just to study how our new approach could overcome such known limitations. In any case, for fairness, we also report data with the PHOLD model configured with no hot-spots, thus naturally leading to a more balanced advancement of virtual time (per wall-clock-time unit) across the LPs, independently of the underlying execution platform among the ones we compare. Finally, in this study we focus on a zero-lookahead scenario.

In Figure 4 we report data related to the configuration with no hot-spot. By the plots we see that SE suffers from performance degradation with respect to USE. In fact, USE allows achieving a maximum speedup—over sequential execution—which is about 34% (resp 18%) better than the one provide by SE for event granularity set to 60 (resp. 120) microseconds. Such a maximum value is achieved for 32 WTs, where USE allows for better exploitation of parallelism, via speculative processing, with respect to SE. We would to note that even if the speculation support in USE is much more lightweight than in traditional speculative PDES, simpler approaches that support a limited level of speculation (e.g., one event for SE) can be more efficient in particular scenarios with a reduced computational power (up to 16 cores) and fine-grain events (60 microseconds); this is why in Figure 4 we note a slightly overcome of SE's performance with respect to USE. In any case, both SE and USE perform better than traditional speculative PDES, namely ROOT-Sim, since they avoid a lot of operations that the traditional engine needs to carry out. As an example, in our USE proposal, the cancellation of events that are no longer valid does not require any anti-event—since it is embedded within the non-blocking event state-machine management in the form of a simple event-buffer state transition. Also, no output queues are generated and traversed for managing rollbacks, since all the work is supported at the level of the SQ where the "positive" copy of the events is posted—still thanks to event-buffers state machines.

When moving to the hot-spot case—whose speedup data are provided in Figure 5—the potential of USE becomes definitely more evident. SE shows performance worse than USE, just because of the impossibility to provide scalable virtual-time synchronization (with no speculation) when hot-spot LPs tend to slow down the advancement of the commit horizon of the simulation. For this workload, USE can provide performance up to 64% (resp. 105%) better than SE for event granularity set to 60 (resp. 120) microseconds. This is a hard-workload scenario which ROOT-Sim cannot cope with in effective manner, even though it implements load-balancing. In

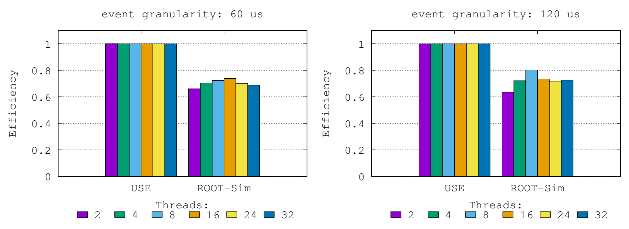


Figure 6: PHOLD efficiency - no hot-spot.

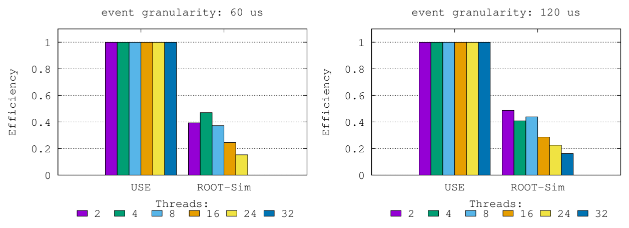


Figure 7: PHOLD efficiency - hot-spot.

fact, with very few spots—10 in our case—long term planning in the distribution of the workload does not capture sudden unbalance, which becomes extremely adverse to performance especially when the number of WTs oversteps the number of hot-spot LPs. On the contrary, USE allows to concentrate the overall computing power towards all the events that are close to the current commit horizon, regardless of their distribution with respect to the hot-spots (or other LPs). As a result, the system gains much more effective parallel execution, with much less likelihood of rollback operations. This phenomenon is evidenced by the efficiency data[3] in Figure 7, where we show that even for this hard workload, USE achieves almost 100% of efficiency, as opposed to ROOT-Sim, which only achieves the order of 40% or less. In Figure 6 ROOT-Sim performs better in the configuration with no hot-spots where it reaches much higher values of efficiency around 70%, though still significantly lower than USE. This shows the effectiveness of delivering the computational power to the highest priority events.

4.2 Results with TCAR

As the second test-bed application, we used a variant of the Terrain-Covering Ant Robots (TCAR) model presented in [14]. In this model, multiple robots (say agents) are located into a region (the terrain) in order to explore it. TCAR simulations are usually exploited to determine tradeoffs between the number of employed robots, and the latency for exploring the target region, e.g., for rescue purposes. Factors such as the speed of movement (depending on, e.g., environmental conditions or obstacles) can be also considered.

In our implementation of TCAR, the terrain to be explored is represented as an undirected graph, therefore a robot is able to move from one space region to another in both directions. This mapping is created by imposing a specific grid on the space regions. The robots are then required to visit the entire space (i.e., cover the whole graph) by visiting each cell (i.e., graph node) once or

[3]We recall that the efficiency of a speculative PDES run is the ratio between the number of committed events, and the total number of processed events, namely committed plus rolled back.

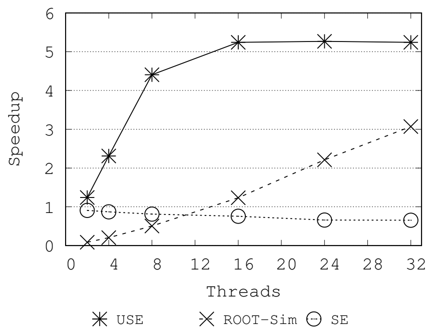


Figure 8: Speedup results for TCAR.

multiple times. Differently from the original model in [14], we have used hexagonal cells—each one modeled by an LP—rather than squared ones. This allows for a better representation of the robots' mobility featuring real world scenarios since real ant robots (e.g., as physically realized in [21]) have the ability to steer to any direction.

The TCAR model relies on a node-counting algorithm, where each cell is assigned a counter that gets incremented whenever any robot visits it. So, the counter tracks the number of *pheromones* left by ants, to notify other ones of their transit. Whenever a robot reaches a cell, it increments the counter and determines its new destination. Destination choice is a very important factor to efficiently cover the whole region, and to support this choice the trail counter is used. In particular, a greedy approach is used such that, when a robot is in a particular cell, it targets the neighbor with the minimum trail count. A random choice takes place if multiple cells have the same (minimum) trail count.

The original TCAR model adopts a *pull* approach for gathering trail counters from adjacent cells. To provide an optimized implementation for PDES, achieved by reducing the volume of interactions (events) across the LPs, we adopted a *push* approach, relying on a notification event which is used to inform all neighbors of the newly updated trail counter whenever a robot enters a cell. Then, each LP modeling a cell stores in its own simulation state the neighbors' trail-counters values, making them available to compute the destination when simulating the transit of a robot. In the used TCAR configuration, we included the evaluation of a new state value for the cell whenever a robot enters it, so as to mimic the evolution of a given phenomenon within the cells. This computation has been based on a linear combination of exponential functions (like it occurs for example when evaluating fading on wireless communication systems due to environmental conditions). We configured TCAR with 1024 cells, with 15% of the cells initially set to be occupied by one robot.

In Figure 8 we show speedup results with the TCAR model. An important aspect for this application is that it shows significantly finer grain events—of the order of a few microseconds—compared to the used configurations of PHOLD. This stresses the execution of the different tested engines along another dimension, with respect to what done with PHOLD. In such a scenario, the overhead for parallelization that is paid by a traditional engine like ROOT-Sim, including its lower efficiency compared to USE, leads to very limited speedup. The same is true for SE, given its impossibility to carry out Time-Warp style speculative processing and the consequent active waiting, namely spinning behind a lock, for causality re-alignment, that increases contention on the underlying memory subsystem. Instead, USE allows achieving speedup that increases up to 16 threads. It then flattens, indicating somehow that more threads do not longer pay-off for performance reduction. We note that USE reaches its maximum speedup beyond a number of thread resources much less than ROOT-Sim indicating its superior ability to exploit actual model parallelism even when scaling up the amount of resources to more limited extents.

5 CONCLUSIONS

In this article we have presented an innovative design of a share-everything PDES system. It allows exploiting CPU-cores on board of shared-memory machines for carrying out the execution of simulation models while guaranteeing: (1) non-blocking coordination of threads in the access to shared data structures and (2) fully speculative—Time Warp-style—processing of the events. No previous literature proposal in the share-everything class shows these two features in combination. Compared to classical Time-Warp PDES—based on binding of LPs to threads—our proposal allows speculative processing of any individual event along whichever thread. Such a fine grain—event based—sharing scheme allows concentrating the computing power towards unprocessed events that are at any time the closest ones to the commit horizon of the simulation. The advantage is the reduction of the incidence of rollbacks, that together with the high effectiveness of platform level non-blocking algorithms for sharing information across threads leads to extreme runtime effectiveness even under hard workload scenarios, as we have shown via an experimental study.

REFERENCES

[1] R. Ayani. LR-Algorithm: concurrent operations on priority queues. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, SPDP, pages 22–25, Dallas, TX, USA, 1990. IEEE Computer Society.

[2] P. D. Barnes, C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS '13*, pages 327–336, 2013.

[3] C. D. Carothers and R. M. Fujimoto. Efficient execution of Time Warp programs on heterogeneous, NOW platforms. *IEEE Transactions on Parallel and Distributed Systems*, 11(3):299–317, 2000.

[4] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

[5] 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*, PADS, pages 1–9. IEEE Computer Society, 2011.

[6] D. Cingolani, A. Pellegrini, and F. Quaglia. Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. *ACM Trans. Model. Comput. Simul.*, 27(2):11:1–11:26, 2017.

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

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

[9] 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*, PADS, pages 15–26. 2014.

[10] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, volume 2180 of *DISC*, pages 300–314. Springer Berlin/Heidelberg, 2001.

[11] 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*, pages 75–86, 2015.

[12] M. Ianni, R. Marotta, A. Pellegrini, and F. Quaglia. Towards a fully non-blocking share-everything PDES platform. In *Proceedings of the 21st IEEE Internat Symp. on Distributed Simulation and Real Time Applications*, pages 25–32, 2017.

[13] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, 1985.

[14] S. Koenig and Y. Liu. Terrain coverage with ant robots: a simulation study. In *Proceedings of the fifth international Conference on Autonomous agents*, AGENTS, pages 600–607. ACM, 2001.

[15] 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*. 2016.

[16] 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*, pages 15–26, 2017.

[17] A. Pellegrini, S. Peluso, F. Quaglia, and R. Vitali. Transparent speculative parallelization of discrete event simulation applications using global variables. *International Journal of Parallel Programming*, 44(6):1200–1247, 2016.

[18] 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*, pages 105–116. 2014.

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

[20] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[21] J. Svennebring and S. Koenig. Building Terrain-Covering Ant Robots: A Feasibility Study. *Autonomous Robots*, 16(3):313–332, 2004.

[22] B. P. Swenson and G. F. Riley. A New Approach to Zero-Copy Message Passing with Reversible Memory Allocation in Multi-core Architectures. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 44–52, 2012.

[23] The High Performance and Dependable Computing Systems Research Group (HPDCS). ROOT-Sim: The ROme OpTimistic Simulator. https://github.com/HPDCS/ROOT-Sim, 2012.

[24] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 211–220. IEEE Computer Society, jul 2012.

[25] J. Wang, N. B. Abu-Ghazaleh, and D. V. Ponomarev. AIR: application-level interference resilience for PDES on multicore systems. *ACM Trans. Model. Comput. Simul.*, 25(3):19:1–19:25, 2015.

[26] 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.*, 25(6):1574–1584, 2014.