ORCHESTRA: An Asynchronous Wait-Free Distributed GVT Algorithm

Tommaso Tocci[†*], Alessandro Pellegrini[*], Francesco Quaglia[¶], Josep Casanovas-García[†‡], and Toyotaro Suzumura[†§]
tommaso.tocci@bsc.es, pellegrini@dis.uniroma1.it, francesco.quaglia@uniroma2.it,
josep.casanovas@bsc.es, suzumura@acm.org
[*]*Sapienza, University of Rome, Italy*
[¶]*University of Rome "Tor Vergata", Italy*
[†]*Barcelona Supercomputing Center (BSC), Spain*
[‡]*Universitat Politècnica de Catalunya (UPC), Spain*
[§]*IBM T.J. Watson Research Center, NY, USA*

***Abstract*—Taking advantage of computing capabilities offered by modern parallel and distributed architectures is fundamental to run large-scale simulation models based on the Parallel Discrete Event Simulation (PDES) paradigm. By relying on this computing organization, it is possible to effectively overcome both the power and the memory wall, which are core limiting aspects to deliver high-performance simulations. This is even more the case when relying on the speculative Time Warp synchronization protocol, which could be particularly memory greedy. At the same time, some form of coordination, such as the computation of the Global Virtual Time (GVT), is required by Time Warp Systems. These coordination points could easily become the bottleneck of large-scale simulations, hindering an efficient exploitation of the computing power offered by large supercomputing facilities. In this paper we present ORCHESTRA, a coordination algorithm which is both wait-free and asynchronous. The nature of this algorithm allows any computing node to carry on simulation activities while the global agreement is reached, thus offering an effective building block to achieve scalable PDES. We claim that the general organization of ORCHESTRA could be adopted by different high-performance computing applications, thus paving the way to a more effective usage of modern computing infrastructures.**

I. Introduction

A core aspect of PDES systems [1] is synchronization, which ensures causally-consistent (i.e. timestamp-ordered) execution of simulation events at each Logical Process (LP). Several synchronization protocols have been proposed, among which the optimism-oriented ones—such as Time Warp [2]—are a viable solution to tackle simulation performance aspects. In Time Warp, events are processed speculatively (thus exploiting parallelism significantly), and causal consistency is guaranteed through rollback/recovery techniques, which restore the simulation model to a correct state upon the a-posteriori detection of consistency violations. This is done via anti-messages (carrying anti-events), which annihilate the originally-sent events, thus possibly causing cascading rollbacks across chains of LPs.

The high level of parallelism of the LPs in their advancement in simulation time is the key to high performance and scalability of model execution. In fact, this execution model tries to capture time independence, without any manual intervention from the simulation model developer. Nevertheless, Time Warp has the need for a global notion of time. A core abstraction is the Global Virtual Time (GVT), which is defined as the smallest timestamp among events (or anti-events) that are still unprocessed, or that are currently being processed. The GVT allows to identify the commitment horizon of the speculative simulation run—no LP can ever rollback to simulation time preceding the GVT value [2]. Its value is used both to execute actions that cannot be subject to rollback, such as displaying/inspecting intermediate simulation results [3], [4], and to reclaim memory [5] (the *fossil collection* operation).

To determine the GVT value, some sort of coordination among the computing nodes is required. This coordination can significantly affect the simulation performance, in a way that is directly affected by the organization of the computing environment. In particular, in order to target the transition from petascale to exascale simulations, the recent architectural trend is to rely on larger clusters of parallel computing machines. Indeed, this is the current way to overcome the *power wall* [6] and the *memory wall* [7], which have posed strict limitations on what can be done with off-the-shelf computing systems.

This modern computing system organization demands for a significant re-thinking of traditional synchronization algorithms. In fact, in order to avoid hampering the overall simulation's performance, we claim that a GVT reduction algorithm must provide two orthogonal features, at the same time. First, it must work (at the level of a single node) in a wait-free way [8]: no critical section must be used, among the processors, to compute the local proposal for the GVT value. Second, at a global scale, the distributed algorithm to determine, among the different local proposals, the actual GVT value must be run in a completely asynchronous way. These properties ensure that modern computing systems used to run simulations based on the PDES paradigm do not suffer from possible performance bottlenecks.

To ensure the correctness of the computed value in a general distributed setting, in-transit messages must be taken into account. This is a non-trivial problem, as the communication network is often such that it is not possible to know automatically what events are still in transit across two different nodes of the network—this is a so-called *non-*

978-1-5386-4028-9/17/$31.00

observable Time Warp system [9]. As already noted in [10], among all the GVT protocols proposed in the literature, those relying on explicit message acknowledgement to cope with in-transit messages limit simulation scalability in a non-negligible way, and must therefore be avoided in large-scale scenarios. Similarly, protocols which do not rely on acknowledgements but use vector clocks to account for in-transit messages have scaling capabilities which are not well-suited for our reference computing environment. On the other hand, algorithms which are based on two cuts (see, e.g., [11]), can be regarded as a viable solution.

In this paper, we propose ORCHESTRA, an algorithm to compute the GVT value in a non-observable distributed Time Warp system, which is at the same time wait-free and asynchronous. Differently from many proposals in the literature, we do not explicitly require FIFO ordering across messages exchanged on the network. ORCHESTRA belongs to the family of two-cuts algorithms, although this is (to the best of our knowledge), the first algorithm to require a single communication round to build the second cut. As it will be shown experimentally, ORCHESTRA is able to deliver significant performance speedups in scenarios where the rest of the simulation environment is already very well optimized. Moreover, the organization of ORCHESTRA is quite general, and can be easily re-adapted to different high-performance computing scenarios.

We have implemented our GVT algorithm into the open source ROOT-Sim speculative PDES platform [12], exactly based on the Time Warp paradigm, and we have also carried out an experimental evaluation to assess the effectiveness of the algorithm on the MareNostrum III supercomputer, using up to 512 CPU cores hosted on 32 computing nodes.

The remainder of this article is structured as follows. Section II discusses related work. ORCHESTRA is presented in Section III. Section IV illustrates experimental results.

II. Related Work

In the literature, a large number of algorithms to compute the GVT value have been proposed. GVT algorithms can be barely classified depending on the specific features of the underlying platform (e.g. shared vs distributed memory) hosting the Time Warp system, and/or on the way they account for in-transit messages in distributed environments.

For shared memory platforms, there are two reference proposals [9], [13]. The former exploits the *observability* property, commonly matched by shared memory implementations of Time Warp, to provide a GVT algorithm that does not rely on any message acknowledgment scheme. This proposal requires synchronization across concurrent processes, materialized in the form of critical sections. The latter relies on the same property, but puts in place a wait-free algorithm, which does not require any critical section to compute the local GVT value. We rely on a variant of this latter proposal, at the level of a single computing node, to determine the local GVT value proposal.

In the context of distributed computing environments, several proposals have been based on explicit message acknowledgment schemes [14]–[18] to determine which messages are still in transit, and which processes are responsible for keeping into account their timestamps while computing the new GVT value. Some of these algorithms (e.g., [15], [18]) acknowledge individual messages, reducing the time interval along which a message can result as still in-transit. Other approaches (e.g., [16]) acknowledge batches of messages reducing the network overhead, but stretching the interval of time along which a message still results in-transit (although being potentially already processed at the destination). This, in its turn, leads to worsening the approximation provided by the algorithms on the actual GVT, given that "obsolete" timestamps might be still considered in the global reduction while computing the new GVT value.

An approach where explicit message acknowledgments are not required has been provided in [11]. In this solution, messages are associated with phases (represented by different message coloring schemes) so that it is possible for the processes in the system to determine whether the timestamp of any message (or anti-message) needs to be accounted for in the current GVT computation. However, this algorithm requires control messages to set-up the start of a new GVT computation. Our proposal belongs to the same family of this proposal, although requiring a smaller number of communication steps to compute the final GVT value, and without any need for communicating the need to start a GVT computation, thus being completely asynchronous.

The work in [19] leverages on both the notion of phases and acknowledgement messages, using a set of parameters to determine the time duration of the phases and to build a sort of "phase groups", in the attempt to increase the timeliness of the GVT reduction. This proposal anyhow requires a very large amount of metadata, and incurs in a non-minimal communication overhead, thus making it non-viable for large-scale distributed simulations.

The need for both control messages and acknowledgments is removed in [20], which has been tailored to distributed memory clusters where specific bounds can be assumed on the message delivery transfer across the nodes, and the clocks of the different machines can be assumed to be (perfectly) synchronized. In this proposal, new GVT computations are triggered by specific timeouts, which occur in synchronized way across all the nodes in the system. We target a more general scenario, where it could be impossible to determine such bounds.

Some proposals [10], [21] try to address a distributed computing organization similar to our reference computing model. The main difference with our proposal is that these works implement a non-homogeneous GVT protocol, where there is some master process which does not carry on any computation, rather acts as an orchestrator of the distributed GVT algorithm. On the other hand, ORCHESTRA is a perfectly-uniform protocol, where there is no global notion of a master process, and where no dedicated processor is destined to the management of the GVT computation.

III. ORCHESTRA

A. Baseline description

ORCHESTRA targets parallel and distributed discrete-event simulations supported by a set of (possibly non-homogeneous) processing units, scattered across any number of machines (i.e., computing nodes). On each computing node, any number of *simulation kernel instances* can be running. These instances are developed according to the symmetric multi-threaded paradigm, as introduced in [22], where shared memory is used to support intra-kernel synchronization. Distributed communication is supported by some network interconnection.

In the most general setting, our reference system model is made up of the following elements:

- A number K of simulation kernel instances (forming up the $KernelSet$), which are scattered across the available computing nodes.
- Each simulation kernel instance $k \in KernelSet$ runs a set a concurrent worker threads, denoted as $TSet_k$. These worker threads rely on shared memory for their internal communication and synchronization tasks.
- At any wall-clock time instant, a worker thread $t \in TSet_k$ is in charge of scheduling events to a set of bound LPs, denoted as $LPSet_t$. As mentioned before, one LP is managed only by one worker thread. Therefore, $LPSet_i \cap LPSet_j = \emptyset \quad \forall i, j \quad i \neq j$.

The overall sequence of messages and events which compose the ORCHESTRA algorithm is shown in Figure 1[1]. ORCHESTRA belongs to the family of two-cut distributed GVT algorithms, and relies on the notion of phases to let the global computation of the GVT value advance, without any form of explicit synchronization. A kernel instance can decide independently from any other to start a GVT computation (i.e., moving from the *idle* phase to the *start* phase), thus avoiding any initial form of communication. Eventually, all distributed instances will move to the *start* phase, collaborating to determine the new GVT value.

At the level of the single kernel instance k, ORCHESTRA relies on phases which are governed by relying on a set of atomic counters. In particular, during the computation of the GVT, every worker thread in the $TSet_k$ set carries on reduction actions on the LPs bound to it, in a way similar to the proposal in [13]. Once a portion of the computation is carried out, each worker thread notifies its completion by atomically decrementing a counter in shared memory, and enters the next phase after all the threads have concluded the current one—this can be done by simply checking the value of a shared counter.

Phase changes at the level of kernel instances are triggered by some global (i.e., distributed) event. In particular, each simulation kernel instance in $KernelSet$ transits through a number of *kernel phases*. The succession of these phases is governed by two different factors. On the one hand, each kernel instance maintains a set of counters to determine when some global condition is met. On the other hand, the completion of some asynchronous collective communication primitive determines the advancement to a different phase.

As for the conditions checked by relying on counters, ORCHESTRA inherits from the proposal in [11] the notion of *colored messages*. In particular, worker threads in ORCHESTRA continuously alternate their execution in a red and a white color. Messages sent while running in the red phase are colored red, and messages sent running in the white phase are colored white. Each worker thread switches from the white to the red phase independently of each other. This means that, at the same wall-clock time instant, two different worker threads (possibly in two different kernel instances) can live in a completely different phase, as depicted in Figure 1. There are only two phase-changing points which are not allowed to be inverted. These are marked by a purple vertical bar in Figure 1. Anyhow, we emphasize that this behavior is not supported by any form of explicit synchronization. In fact, these bars are associated with the completion of asynchronous collective communication primitives. We note that by relying on this scheme, every worker thread on any kernel instance is allowed to carry on simulation work while the GVT value is being computed, thus allowing for an efficient usage of the available computing resources.

As a last note, each kernel instance maintains a counter to identify the current *era value*. This value is used to discriminate between messages sent across two consecutive white phases. In this way, we are able to reduce the amount of metadata exchanged across different kernel instances.

Throughout the description of the various procedures of ORCHESTRA, we rely extensively on the ONLYONCE pseudo-code statement. This statement represents a block of code that should be executed only once by any of the worker threads which concur in the activities of a simulation kernel instance. In practice, we propose to implement such a statement by relying on the Compare and Swap (`CAS`) construct. In particular, for each ONLYONCE statement, an integer *token variable* should be declared. All token variables should be initialized to zero, and every worker thread will try to execute a `CAS(&token, 1, 0)`, meaning that all threads passing through the ONLYONCE statement will try to update the value of `token` to 1. By the semantics of the CAS construct, only one thread will be able to successfully perform the update. Therefore, by checking if the `CAS` has succeeded, only one thread will actually perform the actions associated with this statement.

B. The Algorithm

In a non-observable Time Warp system, there is a time window along which a message associated with a send operation at some LP is considered as *in-transit*, and therefore it has not yet been incorporated into the recipient LP's message queue. To cope with in-transit messages, every time a new event is injected into the system targeting a remote LP, the sender simulation kernel tracks the boundaries

[1]This illustration has given the "ORCHESTRA" name to the algorithm, due to it resembling a music score.

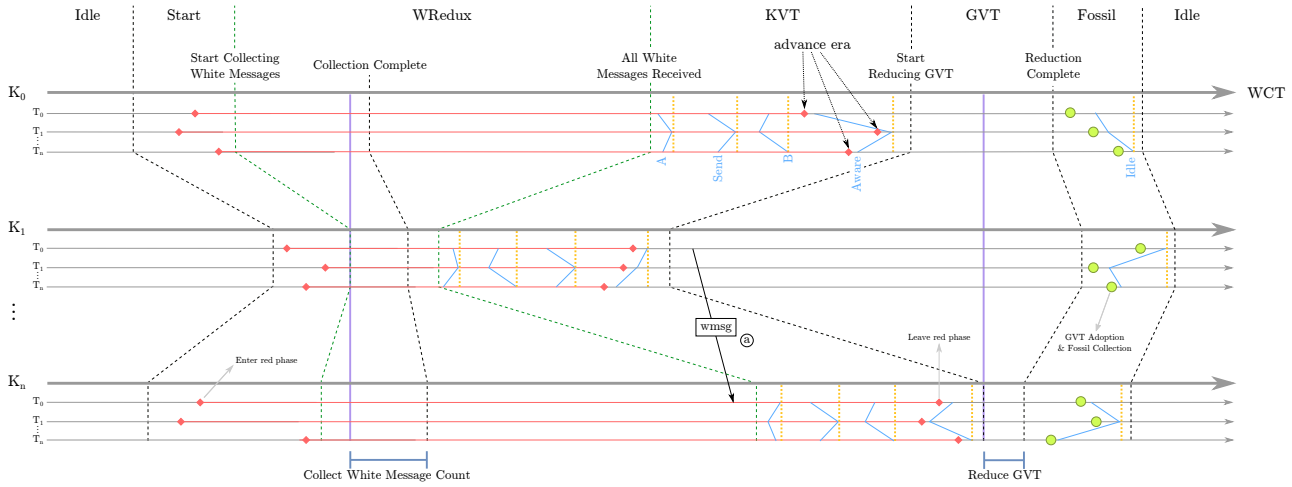

Figure 1. Illustration of the ORCHESTRA algorithm

Algorithm 1 Register Outgoing Event

```
1: procedure RegisterOutgoing(event e)
2:     if e.recipient is locally hosted then
3:         return
4:     if e.colour = red then
5:         min_t^RED ← min{e.timestamp, min_t^RED}        ▷ R1
6:     else
7:         AtomicInc(w_counter_sent[e.kern_dest])
```

Algorithm 2 Register Incoming Event

```
1: procedure RegisterIncoming(event e)
2:     if e.recipient is locally hosted then
3:         return
4:     if e.colour = white then
5:         AtomicInc(w_counter_recv[e.era % 2])           ▷ R2
```

of the non-observability window of the message carrying the event by relying on the RegisterOutgoing() and RegisterIncoming() procedures, executed at the source and destination simulation kernels, respectively. The pseudo-code of these procedures is depicted in Algorithms 1 and 2. As we have discussed, our asynchronous algorithm relies on colored messages to differentiate among the different execution phases of the simulation run in which there is the need to explicitly account for in-transit messages. This is reflected by the fact that the source kernel tracks the minimum timestamp of events scheduled by any LP in the LP sets of its worker threads.

Periodically, a kernel instance determines (independently of the others) that a global agreement on the value of the GVT is to be made, starting the flow across the different phases, which are associated with different states of threads and kernels. We start by illustrating the procedure which is used to compute a local estimation of the GVT value at a certain kernel, which we refer to as the *Kernel Virtual Time* (KVT). The KVT is computed by following the same definition of the GVT, yet by considering only correctly-executed events and in-transit messages which are related to the workers of the specific simulation kernel instance. The KVT computation is carried out in a way similar to what has been presented in [13], and the pseudo-code of the KVT() procedure is presented in Algorithm 3. In particular, when the simulation kernel has entered a GVT-computation phase, it repeatedly invokes the KVT() procedure. The goal of this procedure is to pass through all the required phases (associated with all the worker threads) which ensure a correct estimation of the KVT at the local kernel, returning `true` only when the correct KVT value has been finally computed (**K7**). Once the local KVT computation is started, all the worker threads have been already set into the A phase. In this phase, all threads execute the following actions: i) messages being received from remote kernels are extracted from the underlying communication channel; ii) the events associated with these messages are incorporated into the event queues; iii) the minimum LVT across all LPs bound to each thread is found, and stored into a per-thread variable (**K1**). Once the minimum has been computed, each thread moves to the Send phase, and notifies that it has completed its A phase via the atomic counter C_A (**K2**). This counter ensures that no worker thread will be ever executing the actions associated with the Send phase, until all threads have completed the execution of the A phase (**K3**). We note that the different worker threads are fully allowed to complete the execution of phase A at different wall-clock time instants. This is illustrated by the skewed blue lines in Figure 1. The vertical dotted lines illustrate that the next phase is never started until all threads have completed the previous phase. This algorithmic organization ensures wait-freedom of execution across the different threads.

The steps associated with the Send phase entail the following actions: i) messages are received from the communication channel and incorporated into the event queues; ii) the next event (in a Lowest-Timestamp First fashion) is executed by the worker thread; iii) possibly-generated messages/antimessages are sent. These actions ensure that if some LP bound to a worker thread has executed a rollback

Algorithm 3 Kernel Virtual Time—KVT

```
1: procedure KVT( ): returns boolean
2:     if th_phase = A then
3:         Receive messages from remote kernels
4:         Incorporate messages into event queues
5:         min_t^A ← min_{i∈LPSet_t} LVT_i                          ▷ K1
6:         th_phase ← Send
7:         AtomicDec(C_A)                                           ▷ K2
8:         return false
9:     if th_phase = Send ∧ C_A = 0 then                            ▷ K3
10:        Receive messages from remote kernels
11:        Incorporate messages into event queues
12:        Execute the next event
13:        Send output messages/anti-messages
14:        th_phase ← B
15:        AtomicDec(C_Send)
16:        return false
17:    if th_phase = B ∧ C_Send = 0 then
18:        Receive messages from remote kernels
19:        Incorporate messages into event queues
20:        min_t^B ← min_{i∈LPSet_t} LVT_i
21:        th_current_era ← th_current_era + 1                      ▷ K4
22:        th_colour ← white
23:        min_t ← min{min_t^A, min_t^B, min_t^RED}                 ▷ K5
24:        th_phase ← Aware
25:        AtomicDec(C_B)
26:        if C_B = 0 then
27:            OnlyOnce:
28:                min_k ← min_{t∈TSet_k} min_t                     ▷ K6
29:                return true                                      ▷ K7
30:    return false
```

operation due to a straggler message received during phase A, the LVT of that LP is realigned to that of the straggler message. For a thorough discussion on the correctness of this approach, we refer the reader to [13]. Similarly to the previous phase, all threads switch to the B phase, but they do not start the actions associated with it until all threads have actually completed the Send phase.

At this point, the KVT() procedure shows some differences from the algorithm in [13]. In particular, after having received and incorporated the messages/events, a worker thread can consider the red phase as concluded, switching to the next era as well (**K4**), and computes again its minimum LVT taking into account as well the minimum timestamp of the red messages (**K5**). We note, that switching to the next era does not require any synchronization among the worker threads (as depicted as well in Figure 1), as the correctness of the algorithm is ensured by the consecutive flow of white/red phases. At the same time, once a thread enters the white phase associated with the next era, this white phase can be regarded as a completely-different phase. With respect to Figure 1, let us consider the white message ⓐ sent from T_0 at K_1 to T_0 at K_n. It crosses the "equivalent" second cut of the algorithm in [11]. Since our algorithm does not require FIFO network ordering, this would break the correctness of the algorithm in [11]. Differently, in our algorithm, the correctness is ensured by the fact that the next era virtually bounds the white message ⓐ to the next round of red messages, as depicted in Algorithm 2 (**R2**).

All threads then enter the Aware phase, telling that they know that their contribution to the KVT computation is over.

Algorithm 4 Global Virtual Time—GVT

```
1: procedure GVT( )
2:     if ker_phase = Start ∧ th_phase = Idle then                  ▷ G1
3:         min_t ← ∞
4:         min_t^RED ← ∞
5:         th_colour ← red
6:         th_phase ← A
7:         AtomicDec(C_Init)                                        ▷ G2
8:         if C_Init = 0 then
9:             OnlyOnce:
10:                WMsgsRedux( )                                    ▷ G3
11:                ker_phase ← WRedux
12:        return
13:    if ker_phase = WRedux then
14:        if WMsgsReduxCompl( ) ∧ WMsgsRecv( ) then                ▷ G4
15:            OnlyOnce:
16:                ∀i ∈ KSet: AtomicSet(w_counter_sent[i], 0)
17:                ker_phase ← KVT
18:        return
19:    if ker_phase = KVT ∧ th_phase ≠ Aware then
20:        if KVT( ) then
21:            GVTRedux( )                                          ▷ G5
22:            ker_phase ← GVT
23:        return
24:    if ker_phase = GVT ∧ GVTReduxCompl( ) then                   ▷ G6
25:        OnlyOnce:
26:            new_gvt ← last_reduced_gvt
27:            ker_phase ← Fossil
28:        return
29:    if ker_phase = Fossil ∧ th_phase = Aware then
30:        FossilCollection( )
31:        th_phase ← Idle
32:        AtomicDec(C_End)                                         ▷ G7
33:        if C_End = 0 then
34:            OnlyOnce:
35:                ker_phase ← Idle
```

Once all the threads are in the Aware phase, the global minimum among all the worker threads at the given kernel, namely the KVT value, can be computed as the minimum among the minima of each worker thread (**K6**).

We note that after the execution of every phase, the execution of the KVT() procedure is explicitly interrupted via a **return** statement. This prevents any thread from possibly executing two different phases consecutively. While this would not hamper the correctness of the algorithm, we note that this gives higher priority to the execution of simulation events, which ensures higher performance by reducing the overall rollback probability. The algorithm to compute the KVT value is used in an asynchronously-coordinated manner to compute the GVT value. As mentioned previously, once a certain condition is met, all the simulation kernel instances start executing at every main loop iteration the procedure illustrated in Algorithm 4.

A simulation kernel starts its portion of the GVT computation in the Start phase. Similarly, all its threads are in the Idle phase. In this case (**G1**), we set to ∞ the local min_t and min_t^{RED} values, which will be used in the KVT algorithm to compute the minimum accounting as well for in-transit messages. We then color the running thread red, so that all the messages it will be sending from now on will be colored red as well. Finally, during the initialization phase, we move the thread currently running this procedure into the A phase—since these are symmetric kernel instances,

all worker threads run the same code, and will eventually all enter the A phase. As previously illustrated in Algorithm 3, this phase is associated with the computation of the local (per thread) minimum. Nevertheless, this is only a preparation towards that phase, as it will not affect the execution flow until the KVT() procedure is called, which will happen in a future phase. Similarly to what we have done in the Algorithm 3 (**K2**), we rely on atomic counters (**G2**) to determine when all threads have finished executing the tasks associated with this phase. At the end, by still relying on the ONLYONCE construct, we start collecting all white messages (**G3**). As mentioned before, this is an asynchronous task: each kernel receives the total number of white messages sent by the other kernels, and starts counting all white messages which are received.

The completion of this asynchronous task takes place in the WRedux phase (**G4**): when the total number of white messages is received—this is checked via the WMSGREDUXCOMPL() call—and the counter of in-transit white messages is zero—this is checked via the WMSGRECV() call—the global GVT computation can advance to the next phase. Again, only one thread will force the advancement to the next phase, after having set the white counter to zero for all threads in the local kernel instance. We note that after this phase all in-transit messages have been incorporated into the message queues (possibly causing rollbacks), and therefore all threads can observe the relevant information for the GVT computation locally.

Therefore, the KVT computation as depicted in Algorithm 3 can be repeatedly invoked at every simulation loop. As shown, the KVT() procedure returns true after that all the threads on a simulation kernel instance have flown through all the phases, and the local minimum for one simulation kernel instance has been computed. At this point, another asynchronous global computation can take place, namely the global GVT reduction (**G5**). This phase is semantically equivalent to computing $new\_gvt \leftarrow \min_{i \in KSet} min_i$. Again, this is done by relying on asynchronous calls, and therefore once all kernels agree on the global minimum (**G6**), one single thread will make the kernel advance to the next phase, namely the Fossil phase.

The Fossil phase allows all the worker threads to execute the fossil collection phase on the LPs currently bound to them. While in theory the GVT computation might be considered as already completed, this phase ensures that, independently of the condition which triggers the activation of the GVT computation, a new asynchronous wait-free computation of the GVT value will never be started before the fossil collection is completed. This has the benefit of significantly simplifying the overall structure of the algorithm, making it suitable for most simulation engines without any need to check for critical races on data structures. This is again done by relying on one atomic counter (**G7**). Once this atomic counter is set to zero, one single thread will set the kernel phase to Idle, thus allowing the next GVT computation to take place, whenever the condition is met. At this point all threads are already in the Idle phase, thus the initial conditions of the algorithm have been restored.

To initiate the GVT computation, we restore all the atomic counters used in the GVT() and KVT() procedures to the number of threads locally hosted by the kernel, thus allowing the synchronization on zero to take place. Additionally, we set to zero the counter of white messages received in the previous era. Finally, to actually notify all threads in the kernel that they have to participate in the GVT calculation, the kernel is moved to phase Start, causing the procedure GVT() to actually perform the computation (**G1** in Algorithm 4). The initialization procedure should reset as well all token variables used to implement the ONLYONCE statement.

IV. EXPERIMENTAL RESULTS

We have implemented ORCHESTRA within ROOT-Sim.We have executed experiments by running our implementation on the MareNostrum supercomputer, which is overall composed of 3,056 IBM DataPlex DX360M4 compute nodes, each one equipped with 16 Intel Sandy Bridge cores running at 2.6 GHz and 32 GB of RAM. The operating system is SUSE Linux 11 SP3, and for the communication across nodes we have used OpenMPI v2.1.0.

Regarding the benchmark for assessing the effectiveness of ORCHESTRA, we have used the Personal Communication System (PCS) real-world cellular simulation model, which has already been used as a reference benchmark application in several other studies oriented to optimistic PDES. Each LP models a wireless cell managing 1,000 wireless channels to provide coverage to mobile devices in a hexagonal region. The model is high-fidelity in terms of how interference across different channels within a same cell and power management upon call setup/handoff is captured/actuated. Particularly, the application handles power management simulation according to the results in [23]. The application is also highly parameterizable by allowing the recalculation of fading coefficients and actual Signal-to-Interference Ratio (SIR) both on the occurrence of specific events (e.g. the startup of a call) and periodically (so as to account for, e.g., changes of conditions in the coverage area). Also, the inter-arrival of calls to mobile devices in the coverage area can be configured, thus leading to different values for the wireless channels' utilization factor. This, in its turn, affects both memory and CPU demand by the simulation. The interaction across the different LPs takes place upon a handoff of a mobile device involved in an ongoing communication, in which case the wireless channel at the source cell is released, and a new one in the destination cell is attempted to be reserved.

On our experimentation we set the average residual residence time in one cell to the value 5 minutes, while the average call duration was set to 2 minutes. Both these values have been set to follow exponential distributions. Also, we have set the channel utilization factor to 75%, with balanced workload on all the LPs. This settings produce simulation

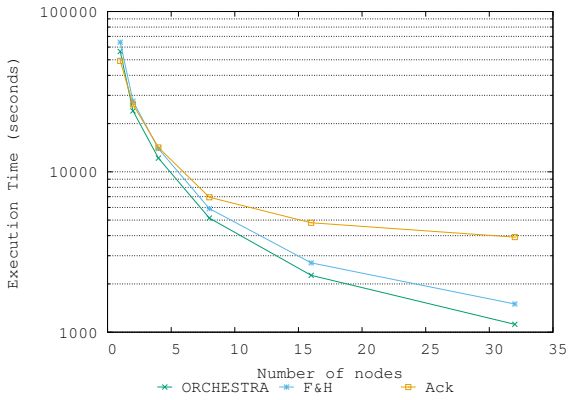

Figure 2. Execution time using 4096 LPs

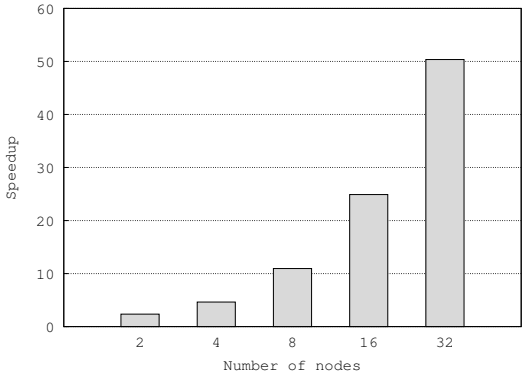

Figure 3. Speedup using 4096 LPs

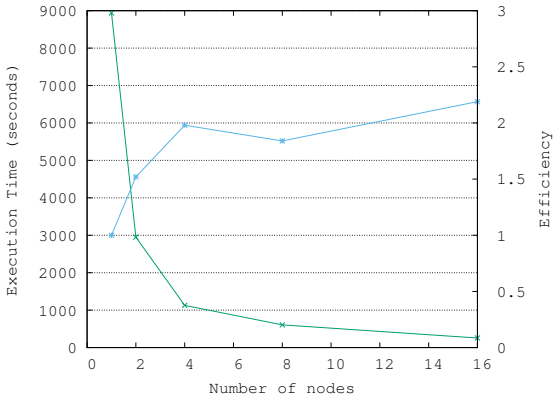

Figure 4. Execution time using 16.348 LPs

event's average CPU requirement of about 150 microseconds. We have run ROOT-Sim on a different number of MareNostrum nodes, namely from 1 node to 32. On every node, we have always used all available 16 cores. We have run two different families of experiments: one using 4096 LPs and running until each cell has completed the simulation of 500,000 calls; one using 16,348 LPs and running until the completion of 10,000 calls.

In Figure 2 we report experimental results when running PCS configured with 4096 LPs. The y axis is logarithmic. Three different GVT reduction algorithms are compared: ORCHESTRA, an asynchronous algorithm where the local computation is protected by critical sections, according to the Fujimoto&Hybinette algorithm in [9] (referred to as F&H in the plot), and an acknowledgement-based reduction inspired to the work in [16] (referred to as Ack in the plot).

With respect to F&H, we observe that ORCHESTRA allows to reduce the wall-clock-time required to complete the run up to 25%. This phenomenon is strictly related to the higher overhead paid by F&H on the local (intra-kernel) computation, in terms of CPU-time required to run tasks related to GVT computation. In fact, since on a single node there are 16 concurrent worker threads active, the likelihood of synchronizing on the GVT-reduction critical section increases. This is explicitly avoided by the phase-based wait-free nature of ORCHESTRA.

A slightly different behavior is observed with respect to Ack. In fact, when running with a small number of distributed nodes (namely, 1 or 2 nodes), Ack is able to deliver a performance which is slightly better, on the order of 12%. This is related to the fact that the communication overhead paid to acknowledge in-transit messages is quite reduced, while the steps required to compute the GVT value are much simpler than in the case of ORCHESTRA. On the other hand, when the degree of concurrency increases, ORCHESTRA is able to deliver a performance increase up to 70%. This is clearly related to the fact that the communication overhead in ORCHESTRA is significantly reduced. This phenomenon is confirmed by the results in Figure 3, where we report the speedup obtained when increasing the number of distributed computing nodes, with respect to the performance obtained when using one single computing node. As it can be seen, when the number of distributed nodes is higher than 2, we observe a super-linear speedup, thanks to the increased overall size of caches across the distributed nodes.

Figure 4 reports data related to ORCHESTRA's scalability when running with a much more increased workload. In particular, the plot shows the simulation execution time when PCS is configured to run 16,348 LPs. By the results, we observe a maximum speedup of 35, when running with 16 nodes, and a trend which is comparable to the one shown in Figure 2, denoting that ORCHESTRA's performance behavior is resilient to a non-minimal scaling up of the workload. The plot shows as well the parallel efficiency of the distributed execution, namely $speedup/n$, where n is the number of distributed nodes. By the results, we can see that ORCHESTRA has a *strong scaling behaviour*, as the efficiency increases without the need for increasing the size of the simulation model.

V. Conclusions

In this paper we have presented ORCHESTRA, an asynchronous wait-free algorithm to compute the GVT value on large clusters of parallel computing machines. ORCHES-

TRA has strong scaling capabilities thanks to the fact that at the level of a single node, concurrent worker threads synchronize in a wait-free fashion. At the same time, the global synchronization is based on a two-cut algorithm, implemented in a completely asynchronous way, which requires a single round of communication to compute the second cut. ORCHESTRA exhibits a super linear speedup, emphasizing the importance of relying on wait-free synchronization and asynchronous coordination with minimal metadata exchange, to efficiently exploit the computing power of modern large-scale supercomputing facilities.

ACKNOWLEDGEMENTS

This work has been partially supported by the Core Research for Evolutional Science and Technology (CREST) program of the Japan Science and Technology Agency (JST).

REFERENCES

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

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

[3] F. Antonacci, A. Pellegrini, and F. Quaglia, "Consistent and Efficient Output-Stream Management in Optimistic Simulation Platform," in *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, PADS*, pp. 315–326, ACM, 2013.

[4] D. Cucuzzo, S. D'Alessio, F. Quaglia, and P. Romano, "A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation," in *Proceedings of the International Symposium on Distributed Simulation and Real-Time Applications, DS-RT*, pp. 227–234, 2007.

[5] S. R. Das and R. M. Fujimoto, "Adaptive Memory Management and Optimism Control in Time Warp," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 2, pp. 239–271, 1997.

[6] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[7] S. A. McKee, "Reflections on the Memory Wall," in *Proceedings of the First Conference on Computing Frontiers*, p. 162, 2004.

[8] M. P. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991.

[9] R. M. Fujimoto and M. Hybinette, "Computing Global Virtual Time in Shared-Memory Multiprocessors," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 4, pp. 425–446, 1997.

[10] G. G. Chen and B. K. Szymanski, "Time Quantum GVT: A Scalable Computation of the Global Virtual Time in Parallel Discrete Event Simulations," *Scalable Computing: Practice and Experience*, vol. 8, no. 4, 2007.

[11] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423–434, 1993.

[12] A. Pellegrini and F. Quaglia, "The ROme OpTimistic Simulator: A Tutorial (invited tutorial)," in *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, PADABS, LNCS, Springer-Verlag, 2013.

[13] A. Pellegrini and F. Quaglia, "Wait-free Global Virtual Time Computation in Shared Memory Time-Warp Systems," in *Proceedings of the 26th International Conference on Computer Architecture and High Performance Computing*, SBAC-PAD, pp. 9–16, IEEE Computer Society, 2014.

[14] B. Samadi, *Distributed Simulation Algorithms and Performance Analysis*. PhD thesis, Computer Science Department, University of California, Los Angeles, 1985.

[15] T. H. Lai and T. H. Yang, "On Distributed Snapshots," *Information Processing Letters*, vol. 25, pp. 153–158, may 1987.

[16] Y.-B. Lin and E. D. Lazowska, "Determining the Global Virtual Time in a Distributed Simulation," in *Proceedings of the 19th International Conference on Parallel Processing, ICPP*, pp. 201–209, Pennsylvania State University Press, 1990.

[17] S. Bellenot, "Global Virtual Time Algorithms," in *Proceedings of the Multiconference on Distributed Simulation*, pp. 122–127, Society for Computer Simulation, 1990.

[18] G. Tel, *Topics in Distributed Algorithms*. Cambridge University Press, 1991.

[19] Y. Peng, L. Qin, and Q. Yin, "A Scalable GVT Estimation Algorithm for PDES: Using Lower Bound of Event-Bulk-Time," *Mathematical Problems in Engineering*, vol. 2015, 2015.

[20] D. W. Bauer, G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman, "Seven-O'Clock: A New Distributed GVT Algorithm Using Network Atomic Operations," in *Proceedings of the 19th Workshop on Parallel and Distributed Simulation, PADS*, pp. 39–48, IEEE Computer Society, 2005.

[21] Z. Lin and Y. Yao, "An Asynchronous GVT Computing Algorithm in Neuron Time Warp-Multi Thread," in *2015 Winter Simulation Conference, WSC*, pp. 1115–1126, IEEE, 2015.

[22] R. Vitali, A. Pellegrini, F. Quaglia, I. Informatica, and G. Antonio, "Towards Symmetric Multi-threaded Optimistic Simulation Kernels," in *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation, PADS*, pp. 211–220, IEEE Computer Society, 2012.

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