A Non-blocking Global Virtual Time Algorithm with Logarithmic Number of Memory Operations

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

Francesco Quaglia
DICII—Università di Roma Tor Vergata
francesco.quaglia@uniroma2.it

***Abstract*—The increasing diffusion of shared-memory multi-core machines has given rise to a change in the design of Parallel Discrete Event Simulation (PDES) platforms. In particular, the possibility to share large amounts of memory by many worker threads has lead to a boost in the adoption of non-blocking coordination algorithms, which have been proven to offer higher scalability when compared to their blocking counterparts based on critical sections. In this article we present an innovative non-blocking algorithm for computing Global Virtual Time (GVT)—namely, the current commit horizon—in multi-thread PDES engines to be run on top of multi-core machines. Beyond being non-blocking, our proposal has the advantage of providing a logarithmic (rather than linear) number of per-thread memory operations—read/write operations of values involved in the reduction for computing the GVT value—vs the amount of threads participating in the GVT computation. This allows for keeping low the actual CPU time that is required for determining the new GVT value. We compare our algorithm with a literature solution, still based on the non-blocking approach, but entailing a linear number of memory operations, quantifying the advantages from our proposal especially for very large numbers of threads participating in the GVT computation.**

I. Introduction

Parallel Discrete Event Simulation (PDES) is the universally recognized technique for achieving high performance and scalable execution of large and/or complex discrete event models [1]. It is based on partitioning the simulation model into several distinct objects—also known as Logical Processes (LPs)—and on enabling threads to concurrently process simulation events occurring at the objects.

Beyond processing events, the threads running within the PDES environment need to execute kind of housekeeping algorithms (or protocols) in order to guarantee correctness of the overall model execution trajectory. These algorithms can be diverse depending on whether the PDES engine entails the possibility to perform speculative event processing [2] or not. In any case, by the course of time, the real implementation of such algorithms has been significantly impacted by the advent and large diffusion of shared-memory multi-core machines. They give the possibility to run PDES platforms by relying on pure shared-data management paradigms, which has led several of the housekeeping algorithms that were originally conceived as distributed ones to slide toward shared-memory based thread coordination algorithms.

A core housekeeping algorithm for PDES is the one used to compute Global Virtual Time (GVT)—namely, the current *commit horizon* of the simulation run. This is important in order to support, e.g., inspections of the advancement trajectory while executing the model [3]. Further, for speculative (optimistic) PDES, GVT computation is fundamental also by the side of memory management, since determining the advancement of the GVT value along time allows for garbage collecting obsolete data (e.g. checkpoints) that were stored in order to correctly support rollback operations in case of erroneous speculation.

In this article we focus on computing the GVT value in a shared-memory multi-thread speculative PDES environment. In particular, we propose an innovative non-blocking GVT algorithm suited for this kind of environments.

Algorithms belonging to the non-blocking class allow manipulating shared-data without relying on critical sections [4]. Rather, they exploit atomic Read-Modify-Write (RMW) machine instructions—like Compare-and-Swap (CAS) and Fetch&Add (FAD)—to enable threads to atomically change portions of (complex) data structures, or to detect that an attempt to make the change fails because of a concurrent update. In the latter case, the operation can be either retried or pushed toward a fall-back path that possibly enables success [5]. With this paradigm, the execution of any thread is never delayed because of the activities by concurrent threads since no lock-protected critical section is ever in place. This allows for both scalability and resilience to thread reschedule.

As far as we know, a non-blocking algorithm to compute GVT in speculative PDES has been already proposed in the literature [6]. The novelty in the GVT algorithm we present compared to such literature proposal is that we carry out the computation with logarithmic number of local (per-thread) memory operations vs the number of involved threads, rather than a linear one as in [6]. This allows reducing the actual CPU time spent while executing steps of the GVT algorithm especially for largely scaled up thread counts. As an advantage, the saved CPU time can be exploited for carrying out other tasks that are functional to advancing the simulation model execution—such as event processing.

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

We provide experimental results demonstrating a significant reduction of the CPU time usage by our proposal compared to the solution in [6]. Particularly, we observe a reduction of CPU time usage for GVT computation of more than 50% when the thread count approaches the value 16K.

The remainder of this article is structured as follows. In Section II we discuss related work. The non-blocking GVT algorithm with logarithmic number of memory operations is presented in Section III. Experimental results are provided in Section IV.

II. Related Work

Different GVT algorithms have been proposed in the literature, which are suited for different kinds of systems, namely distributed-memory systems vs shared-memory ones.

In distributed-memory systems, the core problem to cope with while computing the GVT value is to account for messages (or anti-messages) that are still in transit across processes. In fact, by its definition (see, e.g., [2]), the GVT value must represent the minimum timestamp of any task (or event) that may still occur in the overall system. In this contexts, a few proposals have been based on explicit message acknowledgment schemes [7]–[9] in order to determine which messages (or anti-messages) are still in transit and which processes are responsible for keeping into account the timestamps of in-transit messages while computing the GVT value. Some of these proposals (see, e.g., [7], [9]) opt for acknowledging individual messages, which reduces the time interval along which a message can result as still in-transit. Other approaches (see, e.g., [8]) opt for acknowledging batches of messages (rather than individual ones), which allows for reducing the overhead due to acknowledgment messages but stretches 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.

The proposal in [10] avoids message acknowledgments. It associates messages 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. On the other hand, this algorithm requires control messages to set-up the start of a new GVT computation.

The need for both control messages and acknowledgments is removed by the proposal in [11], 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 computations of the GVT value are triggered by specific timeouts that occur in synchronized way across all the nodes in the system. This gives rise to the scenario where all the nodes observe the start of the GVT algorithm at the same identical time instant, and are able to determine what messages (or anti-messages) can be still in transit since the start of the current GVT computation, given the knowledge on the upper bound delivery delay.

Still tailored to message-passing systems, the GVT algorithms presented and evaluated in [12] share with our approach the idea of avoiding blocking phases. This enables such proposals to scale to large numbers of CPU-cores in the underlying computing platform, an objective that we target as well. However, such approaches primarily target scalability in relation to messaging operations, while we focus on the reduction of CPU time usage for GVT memory operations when large numbers of threads are involved.

Differently from all the above proposals, we target (speculative) PDES systems to be run on top of shared-memory multi-core machines, rather then distributed-memory clusters.

As for literature results tailored to tightly coupled shared-memory systems, the common reference GVT algorithm is the one in [13]. This algorithm requires the PDES system to be "observable", a property which expresses that no message (or anti-message) can ever be in-transit, given that the corresponding send operation leads to directly incorporating the message into the recipient message-queue. Such property is clearly suited for PDES platforms relying on pure shared-memory implementations of housekeeping protocols such as message (event) exchange across different worker-threads. In this algorithm, the start of the GVT computation phase is instantaneously visible to all the processes, given that it simply requires setting the value of a counter into shared-memory to the number of participating processes. However, the computation of the local minimum at each process and the decrease of the counter in order to indicate that the contribution by the process has been made available, are executed within a critical section, which may represent a major impairment to scalability. In our algorithm we avoid any critical section, by trading-off in a completely different way synchronization costs and the number of phases required to compute the new GVT value in a fully non-blocking fashion.

Similar considerations apply to the shared-memory GVT algorithm presented in [14], which is based on a critical section used to atomically update the entries of an array of elements with size equal to the number of participating processes/threads.

As pointed out before, the only non-blocking GVT algorithm for shared-memory (observable) PDES systems we are aware of is the one presented in [6]. In this algorithm, non-blocking manipulation of atomic counters is used to identify what is the current execution phase of the GVT algorithm at each thread, say whether the thread is still determining its local minimum, or has already determined it

and can therefore post such a value onto an entry of an array for enabling the actual reduction leading to the new GVT value. We share with this solution basic principles in relation to the usage of atomic counters for discriminating phases of the GVT algorithm. However, while the GVT reduction in [6] is carried out as a traversal on the array of local minima (with linear cost) carried out by any thread to select the absolute minimum, in our non-blocking GVT algorithm the computation of the absolute minimum within the array is itself an additional protocol, executed along a sequence of non-blocking phases by having each thread accessing a constant number of memory locations in each phase. Given that the number of phases is the logarithm of the number of entries in the array—which corresponds to the number of threads—we finally achieve logarithmic number of memory operations carried out by a thread for determining the GVT.

We note that, in principle, once the array of local minima is fully populated, the computation of GVT via traversal of the array in the approach [6] could be carried out by a single thread (rather than all). This thread could then make the new GVT available to the others. However, adopting this optimization (in terms of reduction of CPU time spent for traversal operations) as a variation of the scheme in [6] would lead a single thread to (significantly) decelerate its event-processing activities, especially for very large numbers of the entries of the array to be traversed. This is potentially adverse to the generation of rollbacks in speculative executions, since the simulation objects bound to the other threads might be allowed to run significantly ahead in logical time, with increased likelihood of being hit by an event with timestamp in the past. This negative phenomenon might appear especially when executing models with fine-grain events. In our solution we reduce the overall CPU time used for GVT computation compared to the original proposal in [6] while still keeping the CPU usage for the GVT algorithm relatively balanced across threads. In fact, a thread performs at worst $O(log(n))$ memory operations more that another one, rather than $O(n)$, with n being the number of involved threads (or array entries). This can favor the avoidance of the divergence of logical clocks of the objects bound to different threads while performing the non-blocking GVT computation.

Finally, the problem of computing reductions efficiently has been also addressed for the case of SIMD (Single Instruction Multiple Data) architectures, such as in CUDA [15]. In this work the objective is the one of achieving a *work-efficient* parallel solution, say one that does not require more computation steps than the sequential solution. This approach allows performing the reduction (in particular, Parallel Prefix Sum) in a work-efficient manner by relying on a total number of $O(n)$ computing steps, while we enable $O(log(n))$ memory operations to be carried out by each individual thread (so we guarantee logarithmic thread-local work). However, the proposal in [15] is based on fully synchronous execution across the threads, as typical of SIMD approaches, while we enable non-blocking GVT computation on MIMD (Multiple Instructions Multiple Data) platforms thus allowing threads to interleave reduction tasks with other kinds of tasks.

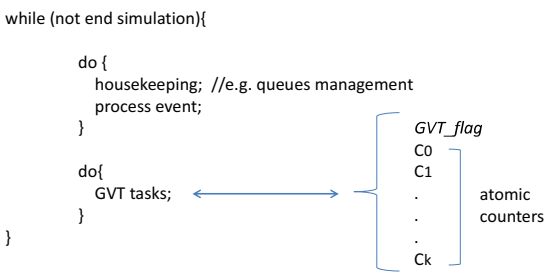

Figure 1. The main loop of a thread - GVT tasks are executed at each loop in non-blocking fashion subject to GVT_flag being true.

III. Non-blocking GVT Computation with Logarithmic Number of Memory Operations

As hinted, our non-blocking GVT algorithm works by having each thread operating within phases. No thread can pass to the next phase until all threads have concluded their tasks in the current phase. However, this is not realized by using blocking synchronization schemes. Rather, we exploit atomic counters as in the spirit of the proposal in [6]. If a thread has already finalized its task in the current phase of the GVT algorithm, it increments the atomic counter associated with such phase and then simply resumes processing other tasks (events and other housekeeping operations)—rather than remaining blocked waiting for the other threads to finalize their corresponding tasks. On the other hand, querying the atomic counters periodically allows the thread to get insights on the fact that all the threads have finalized their tasks related to the current GVT computation phase, so that each of them can move to the next phase.

At a high level of abstraction, the schematization of the execution flow of a thread in our target PDES environment is provided in Figure 1. Essentially, the thread executes a main loop where it performs housekeeping operations—such as the management of the event queues associated with the simulation objects bound to it—and actually processes events. However, at each iteration of the loop, it also executes some non-blocking step of the GVT algorithm.

In particular, the thread checks if GVT computation is currently in place, by checking the GVT_flag in shared-memory. In the positive case, it uses a thread-local variable $current\_phase$ to identify in which phase of the GVT algorithm it currently resides. This will lead the thread to identify what atomic counter C_i needs to be currently exploited to drive the execution of the GVT algorithm. Specifically, as soon as the thread finalizes its task associated with the i-th phase of the non-blocking GVT algorithm, it increases atomically the corresponding counter C_i via a FAD

Algorithm 1 Generic GVT computation phase at thread i

```
if (GVT_flag){
    if (task_{current_phase} not done)
    then{
        do task_{current_phase}; //do the job associated with the current phase
        FAD(C_{current_phase}); //signal via the atomic counter that the job has been done
    }
    if (C_{current_phase} = NUM_THREADS){ //all threads finalized the current phase - move to next phase
        current_phase = current_phase + 1;
    }
}
```

machine instruction. If the counter value after the atomic update is equal to the number of involved threads then it means that entering the next phase is admitted. This will lead the thread to update its $current\_phase$ variable since all threads are allowed to move to the next phase. By the schematization of execution flow in Figure 1, the check on the value of C_i is carried out periodically within the main loop, still in non-blocking fashion by simply reading the current counter value. Algorithm 1 reports the pseudo-code describing the actions that are performed by a thread in order to carry out its computation within the i-th phase of the GVT algorithm, according to the description we have provided.

A few phases of our non-blocking GVT algorithm correspond to the ones in the solution in [6]. In particular, we refer to that article for all the phases related to the identification of the local minimum by each thread. However, as hinted before, in [6] the task associated with a core phase of non-blocking GVT computation corresponds to traversing all the entries of an array where each thread has previously posted its local minimum and in identifying the absolute minimum. We refer to as $task_{reduction}$ the task associated with such a reduction phase. Also, we denote with $values_0[]$ the array where each thread has posted its local minimum.

In our proposal, $task_{reduction}$ is no longer a traversal on the array $values_0[]$. Rather, this task is divided into phases still executed in non-blocking mode and coordinated via atomic counters indicating how many threads already finalized their activity within the current phase.

In the first phase of reduction a thread identifies a competitor thread and makes a challenge with it in order to post its local minimum onto one entry of another array of half of the size of $values_0[]$, which initially has all its values set to $dummy\_value$. We refer to the half size array associated with the first phase of reduction as $values_1[]$. Similarly, $values_i[]$ will denote the array used for the i-th phase of reduction—which will still be based on a challenge between pairs of threads—and we have that the size (number of entries) of $values_i[]$ is always half of the size of $values_{i-1}[]$.

We denote with T_i a thread and with T_j its competitor. In the first phase of reduction, both these threads try to post their local minima, namely $values_0[i]$ and $values_0[j]$, onto the same entry x of $values_1[]$ using a CAS machine instruction. Therefore, only one of the two threads will actually win the race and will have its local minimum loaded onto $values_1[x]$. The winner has concluded its task, and will participate in the other reduction phases by doing nothing—it will simply pass through these phases by still relying on updates and periodic checks of the corresponding atomic counters. On the other hand, the looser needs to check whether its local minimum stored in some entry of $values_0[]$ is lower than $values_1[x]$. In the positive case it updates $values_1[x]$ with its local minimum. In this way $values_1[x]$ will keep the value of a reduction that involved $values_0[i]$ and $values_0[j]$.

When all the entries of $values_1[]$ have been filled, meaning that each pair of competitor threads $< i, j >$ have already been challenged, and the corresponding atomic counter indicates this, the next phase of reduction can start, which will involve a similar challenge between newly defined pairs of competitor threads among those that are still in competition since they have lost a previous challenge. At each challenge we have half of the threads that finish their job and the remaining half that need to carry out a new challenge. Thus the number of reduction phases to implement the original $task_{reduction}$ is $log(n)$, with n being the number of threads. We also recall that a constant number of memory operations are performed by a thread in each phase.

Indicating with $alive_h$ the number of threads still involved in challenges—where $alive_0$ is initially set to $NUM\_THREADS$—we can easily get deterministic association of the indexes i, j (the two competitors) and x (the target entry in the half size array) guaranteeing no overlap of the indexes involved in different challenges. In particular, this can be achieved by relying on the following relations that can be computed individually by any thread in constant time:

$$\begin{aligned} i &\in \left[0, \frac{alive_h}{2} - 1\right] \\ j &= alive_h - 1 - i \\ x &= i \end{aligned} \tag{1}$$

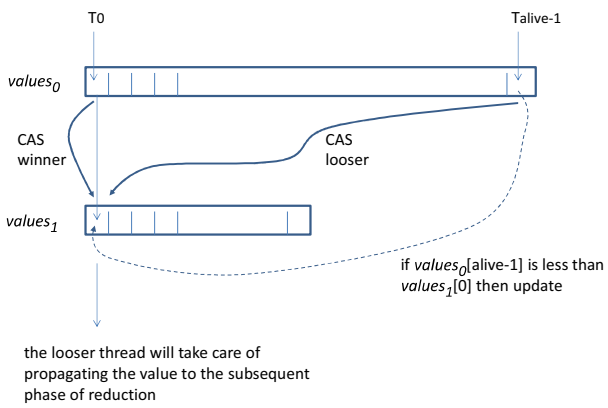

Figure 2. Thread challenge at each phase of reduction.

Also, at the end of the challenge carried out in the h-th phase of reduction, if the j-th thread is the looser, then it sets its logical identity within the GVT algorithm to i for the next iteration, which allows the above equations to still be usable for the $(h+1)$-th non-blocking reduction phase.

An example schematization of the behavior of our decomposition of $task_{reduction}$ into non-blocking phases (or subtasks) is provided in Figure 2. As shown in the picture, and according to Equation 1, T_0 is competitor of T_{alive_0-1}. Also, they both try to update $values_1[0]$ via CAS, with $values_0[0]$ and $values_0[alive_0 - 1]$ respectively.

In this example, T_0 is the winner, so it will simply go ahead passing to the subsequent phases of reduction, with no actual work to be performed. Rather, T_{alive_0-1} checks whether its local minimum kept by $values_0[alive_0 - 1]$ is lower than the current value kept by $values_1[0]$ and if yes it posts its local minimum onto $values_1[0]$. At this point thread T_{alive_0-1} virtually sets its identity to the value 0 so as to participate to the next phase of reduction, specifically figuring as the owner (or the responsible) of $values_1[0]$.

The pseudo-code describing the generic non-blocking phase of such a reduction approach is provided in Algorithm 2. The thread-local variable $winner$ keeps track of whether the thread has won a challenge. Also, the variable $reduction\_phase$ keeps track of the current phase of reduction and the flag $done_{reduction\_phase}$ indicates whether the job to be carried out by the thread in the current phase of reduction has already been completed.

Clearly, as soon as all the phases of reduction are carried out and all the threads enter the $log(n)$-th phase—with n being $NUM\_THREADS$—the corresponding task to carry out simply consists in reading the newly available value of the GVT from $values_{log(n)}[0]$.

As a final note, the start of the GVT algorithm can be simply triggered by setting all the atomic counters to the value 0 and updating GVT_flag to true. This can be done with no risk of critical races by having the threads updating a global variable keeping track of the current GVT round via CAS. The unique thread that updates this variable successfully is the one in charge of triggering the new GVT round by setting the atomic counters and GVT_flag. On the other hand, GVT_flag is reset by the thread that last concludes the execution of the final phase of the non-blocking GVT algorithm according to the scheme proposed in [6]. This is managed by having FAD returning the counter associated with that phase atomically with the update itself. Therefore a thread can always distinguish whether it released the last unit to add to the counter used to track the finalization of the last phase of the GVT algorithm.

IV. Experimental Results

In this section we report experimental results showing the effects of our non-blocking GVT algorithm, comparing it with the proposal in [6] and with a variation of it. This variation is the one we discussed in Section II, in which we enable a single thread to traverse the array where each thread already posted its local minimum—in a previous non-blocking phase of GVT computation according to [6]—in order to finally compute the global minimum. Given that the proposal in [6] has been shown to better scale with respect to blocking algorithms for GVT computation in shared memory systems (such as [13]) and to be more resilient to thread reschedules in terms of performance degradation, we feel it can represent a good reference in our experimental assessment.

Because of the non-blocking nature of all the solutions we compare in this experimental study, we identify the following two main metrics as representative:

- The total CPU time (TOCT) used by the threads for computing a new GVT value. Recall that threads spend their CPU time in parallel and with no dependency due to critical sections (e.g. spin-lock wait phases). Hence TOCT does not correspond to the latency for computing the new GVT. However, given the "background" nature of non-blocking GVT computation in our algorithm and in the competitor approaches we experiment with, CPU time usage appears to be more relevant than latency. This is aligned with the objective of trading-off non-blocking phases vs GVT computation latency to achieve higher scalability, like in the spirit of the approach in [6].
- The maximum difference of CPU-time usage (MDCU) between all pairs of threads participating in the GVT computation.

Given that the CPU time for computing a new GVT value is spent along the critical path of execution of the threads involved in the computation, TOCT is suited to detect whether an algorithm imposes higher or lower overhead to the simulation run. In particular, the lower the value of TOCT the lower the overhead.

On the other hand, MDCU is suited to detect whether some thread is impacted more significantly than the others

Algorithm 2 Generic phase of reduction at thread i - all the reduction phases contribute to execute $task_{reduction}$

if (GVT_flag){
  if ($redution\_phase$ is new **AND** $winner$){ // we entered a new reduction phase - but no work needs to be done since we won a previous challenge
    FAD($C_{reduction\_phase}$); // signal via the atomic counter that the thread finalized this phase
  }

  if ($redution\_phase$ is new **AND Not** $winner$ **AND Not** $done_{reduction\_phase}$){ // we entered a new reduction phase and need to participate
    $outcome$ = CAS($values_{reduction\_phase+1}[x]$, $dummy\_value$, $values_{reduction\_phase}[i]$); // try to update target location via CAS
    if ($outcome$ is FALSE){ // thread lost the challenge
      $values_{reduction\_phase+1}[x] = min(values_{reduction\_phase+1}[x], values_{reduction\_phase}[i])$;
      $done_{reduction\_phase}$ = TRUE;
      set thread identity to x; // it might already correspond to x (see Equation 1) but setting is anyhow safe
    }
    else{
      $winner$ = TRUE;
    }
    FAD($C_{reduction\_phase}$); //signal via the atomic counter that the thread finalized this phase
  }

  if ($C_{reduction\_phase} = NUM\_THREADS$){ //all threads finalized the current phase of reduction - move to next phase
    $reduction\_phase = reduction\_phase + 1$;
  }
}

in terms of the CPU time it spends for participating in the GVT algorithm. In other words, it indicates whether the workload for GVT computation is fairly distributed across all the threads or not.

It appears clear that a low (or very low) value for TOCT does not necessarily represent the ideal outcome since, as we noted in Section II, experiencing non-minimal values of MDCU indicates that most of the work required for GVT computation is executed by a reduced number of threads (or by a single one). In such scenarios, given the non-blocking nature of GVT computation—which still enables speculative processing of events within the PDES system while GVT tasks are executed in background style—the threads that are mostly impacted by GVT computation may generate a situation where the simulation objects they are managing can remain back in logical time. This can happen exactly because less CPU time is devoted to processing the events along the threads they are bound to compared to what happens for simulation objects bound to other threads—those threads that are less impacted by GVT computation. Such temporarily unbalanced usage of CPU for computing the new GVT value may therefore give rise to unbalance in the speculative advancement of the logical time of the simulation objects, a phenomenon that is recognized to be adverse to optimistic synchronization.

Overall, non-minimal MDCU values are related to a *sudden imbalance* caused by GVT computation which is not easily manageable. In fact, it is not directly faceable via common (dynamic) load-balancing algorithms for speculative PDES (e.g. [16]–[18]), which are essentially tailored to keep the execution balanced on the long term just depending on the simulation workload, not the per-thread cost for computing GVT. Overall, the ideal non-blocking GVT algorithm would be the one jointly guaranteeing low TOCT and low MDCU.

Clearly, our non-blocking GVT algorithm is suited for very large numbers of speculative PDES threads (e.g. thousands) running on the underlying multi-core machine in real concurrency. In fact, this is the scenario where the logarithmic memory operations complexity it provides can originate benefits. On the other hand, testing our proposal with thousands of threads in a real deploy of a speculative PDES environment would require the same amount of CPU-cores to be available in the underlying hardware, which at current date is an unlikely scenario except for data centers embedding high-end computing facilities. To bypass this problem, so as to be able to experiment with very large thread counts on top of medium-end off-the-shelf multi-core machines entailing common CPU-core counts—of the order of a few tens—which are available in our laboratory, we have taken the following approach.

We assesses our proposal embedding it within a skeleton that mimics the thread-level dynamics of speculative PDES platforms (say the loop presented in Figure 1), but where the computation in the `do{housekeeping; process event;}` code block is replaced by a blocking operating system sleep. In this way, we can accommodate much larger numbers of threads on the underlying multi-core machine without leading them to use CPU cycles except for GVT computation tasks. This provides also the advantage that we can easily measure the actual usage of CPU time by the threads exploiting the `getrusage()` system call offered by Linux, which is our underlying operating system. In fact, any user-space CPU time accounted for by this service is

actually related to GVT computation tasks since no other activity is carried out in user space code by the threads running the skeleton. We also note that relying on operating system sleeps does not affect the reliability of data we collect, just because the GVT tasks in the algorithms we experiment with are non-blocking. Therefore the CPU time they require along a thread is essentially independent of whether other threads are in the sleep state or not.

In the skeleton, we have implemented the computation of the local minimum to be posted onto the $values_0[]$ array as a simple extraction of a random timestamp. This enables us to assess the various non-blocking GVT algorithms without biasing the study towards a specific implementation of the speculative PDES engine. In fact, different engines may pay different costs to compute the local minimum depending on their internal organization—as an example, the cost can be influenced by whether each thread manages a single queue of events for all its bound simulation objects or separated event queues. In other words, in our study we account for the net CPU time required for running non-blocking GVT tasks provided that the local minimum is already available according to the way the specific PDES engine computes it along a thread.

All the experiments have been carried out running the skeleton on top of a 32-core HP ProLiant machine equipped with 64 GB of RAM, running Linux (kernel version 3.2). We report results achieved by running with up to 16K threads since going beyond this value has generated a freeze of the operating system because of the unavailability of memory for managing the launched threads, and consequently also kernel level demons/operations.

In Figure 3 we report the values we have observed for TOCT for the case of all the non-blocking GVT algorithms we have compared. Each reported value has been computed as the average over 100 GVT computations by each algorithm. We denote with Log-NBGVT our new proposal, with Lin-NBGVT the linear cost solution in [6] and with OL-NBGVT the variation of this algorithm where a single thread traverses the array keeping the local minima for determining the new GVT value. For the reported data set we have used an operating system sleep period of 1 second within the skeleton, but we observed no significant impact on the values of TOCT (and also MDCU) by setting different values of the sleep time. As hinted before, this is expected just because of the non-blocking nature of the tasks carried out in the different GVT algorithms we experimented with.

By the data we see how, for thread counts of the order of 2K and 4K, Log-NBGVT initially shows a slight increase of the total CPU time for computing the GVT value compared to Lin-NBGVT. This is due to the higher cost spent by Log-NBGVT for managing atomic counters, along multiple phases, which does not yet pay-off with respect to the cost spent by Lin-NBGVT for traversing the array of local minima along any thread. However, as soon as the thread

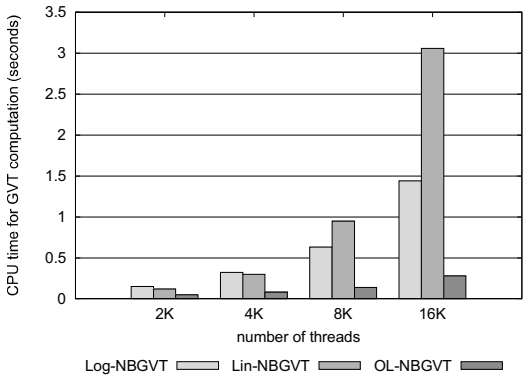

Figure 3. Total CPU time for GVT computation.

count is scaled up to 8K or 16K, Log-NBGVT provides a reduction of TOCT by slightly more than 50%. In fact, for 16K threads, Lin-NBGVT spends around 3 seconds of CPU time (slightly less than 200 microseconds per-thread on the average) for the GVT computation while Log-NBGVT spends around 1.5 seconds of CPU time (100 microseconds per-thread on the average). Better reductions of TOCT area achieved by OL-NBGVT. However, this variation of Lin–NBGVT definitely suffers from the problem of creating an imbalance in the cost spent by the threads for carrying out GVT tasks. This is evident when looking at data related to MDCU which are reported in Figure 4. For thread count scaled up to 16K, we observe a value of MDCU by OL-NBGVT of the order of slightly less than 400 microseconds. This value is even worse than the average per-thread CPU cost spent by Lin-NBGVT under the same setting probably due to the lower effectiveness of caching when a single thread tries to access the array in read mode for performing the traversal operation. In fact, having multiple threads doing this job as in Lin-NBGVT can favor the joint exploitation of lower level caches (e.g. the L3 cache). Ideally, for fine-grain models, say of the order of 30 micro-seconds per event, the other concurrent threads could process (16K-1)$\times$400/30 events, while the simulation objects bound to the unique thread traversing the array of local minima in OL-NBVGT would be locked in logical time. If at least one of these events is eventually impacted—in terms of its causal consistency—when resuming the processing along that thread, a potential cascade of rollbacks might lead to significant wasted computation. Log-NBGVT alleviates the problem since the skew in CPU time usage by all the pairs of different threads is definitely reduced, even with thread count scaled up to 16K. In fact, for such a large number of threads it is of the order of 30 microseconds.

V. Conclusions

In this article we have addressed the problem of computing the GVT value in speculative PDES platforms to be run on top of shared-memory multi-core machines. We have

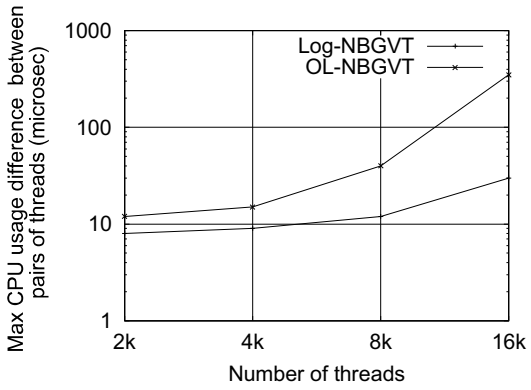

Figure 4. Max CPU-time usage difference between any pair of threads participating in the GVT computation.

presented a non-blocking GVT algorithm such that no thread is ever delayed while carrying out GVT computation tasks depending on the actions by other threads. Logically, the algorithm partitions the computation of a new GVT value into phases that are executed asynchronously by the threads, and where each thread determines that a new phase can be entered by simply checking the value of some atomic counter residing on shared-memory. The number of phases is logarithmic with respect to the number of values to be involved in the reduction for computing the new GVT—which corresponds to the number of participating threads—and in each phase only a constant number of buffers (values) are manipulated by a thread. This allows reducing the actual CPU time required for the reduction compared to a literature non-blocking GVT algorithm where each thread needs to scan an array keeping the input values of the reduction. Also, the save of CPU time is achieved by having it balanced across the participating threads, which is important in order to avoid sudden skews in the advancement of the simulation along the different threads running within the PDES platform. An experimental study with a synthetic testbed where up to 16K threads are run shows that such a CPU save can reach the order of 50%.

REFERENCES

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

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

[3] Diego Cucuzzo, Stefano D'Alessio, Francesco Quaglia, and Paolo Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real Time Applications*, pages 227–234, 2007.

[4] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[5] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. A conflict-resilient lock-free calendar queue for scalable share-everything PDES platforms. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 15–26, 2017.

[6] Alessandro Pellegrini and Francesco Quaglia. Wait-free global virtual time computation in shared memory Time-Warp systems. In *Proceedings of 26th IEEE International Symposium on Computer Architecture and High Performance Computing*, pages 9–16, 2014.

[7] Tel Gerard. *Topics in Distributed Algorithms*. Cambridge University Press, 1991.

[8] Yi-Bing Lin and Edward D. Lazowska. Determining the global virtual time in a distributed simulation. In *Proceedings of the 19th International Conference on Parallel Processing*, pages 201–209, 1990.

[9] Ten-Hwang Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[10] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[11] David W. Bauer, Garrett Yaun, Christopher D. Carothers, Murat Yuksel, and Shivkumar Kalyanaraman. Seven-o'clock: A new distributed GVT algorithm using network atomic operations. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation*, pages 39–48, 2005.

[12] Kalyan S. Perumalla, Alfred J. Park, and Vinod Tipparaju. Discrete event execution with one-sided and two-sided gvt algorithms on 216,000 processor cores. *ACM Transactions on Modeling and Computer Simulation*, 24(3):16:1–16:25, 2014.

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

[14] Zhonge Xiao, Fabian Gomes, Brian Unger, and John G. Cleary. A fast asynchronous gvt algorithm for shared memory multiprocessor architectures. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 203–208, 1995.

[15] M. Harris, S. Sengupta, and J.D. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 3(39):851876, 2007.

[16] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Load sharing for optimistic parallel simulations on multi core machines. *SIGMETRICS Performance Evaluation Review*, 40(3):2–11, 2012.

[17] Christopher D. Carothers and Richard M. Fujimoto. Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 11(3):299–317, 2000.

[18] D. W. Glazer and Carl Tropper. On process migration and load balancing in time warp. *IEEE Transactions Parallel and Distributed Systems*, 4(3):318–327, 1993.