# GVT Algorithms and Discrete Event Dynamics on 129K+ Processor Cores

Kalyan S. Perumalla, Alfred J. Park, Vinod Tipparaju
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
{perumallaks, parkaj, tipparajuv}@ornl.gov

***Abstract*—Parallel discrete event simulation (PDES) represents a class of codes that are challenging to scale to large number of processors due to tight global timestamp-ordering and fine-grained event execution. One of the critical factors in scaling PDES is the efficiency of the underlying global virtual time (GVT) algorithm needed for correctness of parallel execution and speed of progress. Although many GVT algorithms have been proposed previously, few have been proposed for scalable asynchronous execution and none customized to exploit one-sided communication. Moreover, the detailed performance effects of actual GVT algorithm implementations on large platforms are unknown. Here, three major GVT algorithms intended for scalable execution on high-performance systems are studied: (1) a synchronous GVT algorithm that affords ease of implementation, (2) an asynchronous GVT algorithm that is more complex to implement but can relieve blocking latencies, and (3) a variant of the asynchronous GVT algorithm, proposed and studied for the first time here, to exploit one-sided communication in extant supercomputing platforms. Performance results are presented of implementations of these algorithms on up to 129,024 cores of a Cray XT5 system, exercised on a range of parameters: optimistic and conservative synchronization, fine- to medium-grained event computation, synthetic and non-synthetic applications, and different lookahead values. Performance to the tune of tens of billions of events executed per second are registered, exceeding the speeds of any known PDES engine, and showing asynchronous GVT algorithms to outperform state-of-the-art synchronous GVT algorithms. Detailed PDES-specific runtime metrics are presented to further the understanding of tightly-coupled discrete event dynamics on massively parallel platforms.***

***Index Terms*—Parallel Discrete Event Simulation, Time Warp, Global Virtual Time, One-sided Communication, Asynchrony**

## I. Introduction

Parallel discrete event simulation (PDES) [1] is used for simulating large scenario configurations in several important areas such as epidemiological outbreak phenomena, Internet modeling, vehicular transportation, emergency/event planning, and social behavioral simulations, to name a few [2]. Discrete event execution evolves the states of the underlying entities (e.g., vehicles) in an asynchronous fashion, in contrast to time-stepped execution in traditional scientific computing applications in which the entire system state is (logically) updated over fixed time steps.

In PDES, independent logical processes (LPs) hold encapsulated states and evolve their states along a virtual time axis, and exchange timestamped events to incorporate inter-LP data dependencies. In *conservative* PDES, an LP does not execute an event until it can guarantee that no event with a smaller timestamp will later be received by that LP. In *optimistic* PDES, events are potentially executed before such a guarantee can be obtained, but, suitable corrective action (called rollback) is performed on the incorrectly processed events if any timestamp order violation is later discovered. PDES runtime engines may support conservative, optimistic, or both approaches. Runtime engines for discrete event simulations need to deliver fast and accurate global timestamp-ordered execution across a large number of processors, to speed up large-scale scenarios in a range of applications. Among the major challenges in scaling the PDES runtime engines is the design and development of appropriate algorithms that advance the *global virtual time* (GVT) which directly determines the advancement of the distributed wave of progress of all processors executing events staggered along a global virtual timeline.

While multiple equivalent definitions of GVT are possible, here, we shall view GVT as a virtual time value $T_{min}$ such that no processor shall receive any event $E$ with a timestamp $T_E$ such that $T_E < T_{min}$. Thus, each processor, after receiving a value of $T_{min}$, can commit local processing until $T_{min}$ without fear of any data dependency violation. Clearly, the rapidity with with $T_{min}$ can be advanced globally determines the speed with which processors can concurrently execute their event work loads; and, in turn, the faster the increase of the next global minimum time would be. The fine-grained nature of event execution imposes tight constraints on GVT algorithms with respect to scalability. The scalability and efficiency of GVT algorithms can only be properly experimented with actual software implementation and benchmarking using PDES engines and applications at scale.

Here, we look at GVT algorithms that can scale to massively parallel platforms, and focus on three major variants that span the space of synchrony *vs*. asynchrony, and traditional two-sided communication *vs*. newer one-sided communication approaches. With two-sided communication, we propose two

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

algorithms generalized to include optimistic as well as conservative discrete event execution. With one-sided communication, we propose a new asynchronous GVT algorithm that can be mapped directly to the one-sided communication interfaces supported by some of the largest supercomputing systems.

In evaluating the GVT algorithms, we study their performance by varying four different dimensions in PDES applications: (1) event dependency structure, determined by the application's event computation characteristics such as event granularity, and the distribution of timestamps dynamically generated by events, (2) conservative or optimistic synchronization, which determines whether LPs can process some local events beyond GVT, (3) lookahead, which is a measure of static concurrency available in the application scenario, and (4) inter-processor messaging types, categorized here as two-sided and one-sided.

Several global virtual time algorithms have been proposed over the past two decades [1]. In the history of GVT algorithms, a few salient items are: the Mattern's algorithm [3], hardware supported algorithms [4], [5], centralized algorithms [6], unreliable and reliable network-based algorithms [7], and reductions-based algorithms [7], [8]. Few GVT algorithms have been gainfully employed on supercomputers with 100,000+ processor cores, and relatively little is known on the dynamics of discrete execution on a range of representative applications and benchmarks. Even fewer focused on the potential to exploit one-sided communication in GVT computation. Although one-sided communication has been used in other parallel applications, its use in PDES is different in that it mixes the fixed structures and data volumes of GVT messaging with the dynamic structures and data volumes of the inter-processor event exchanges. PDES, as is well-known, is highly latency-bound, making it an excellent class of applications that can potentially exploit the benefits of fast one-sided communication.

The rest of the paper is organized as follows. The GVT algorithms are described in Section II, and their implementation details are presented in Section III. A detailed performance study on a variety of PDES benchmarks is described in Section IV, followed by a summary in Section V.

## II. Synchronous and Asynchronous GVT

In a typical discrete event execution, the execution engine operates in a loop of processing local events (main loop), and participates in inter-processor synchronization for GVT. Depending on the specific needs of the synchronization scheme employed by the engine, a GVT computation is initiated inside the main loop. For example, a conservative engine initiates a new GVT when it runs out of local events to process safely. An optimistic execution initiates either at a predefined frequency or on demand when memory used for rollback support needs to be reclaimed. In the generalized hybrid engine, $\mu sik$ [9], that supports both conservative, optimistic, and mixed synchronization, GVT is always initiated as soon as a previous GVT completes, to minimize blocking for conservative LPs, and to minimize uncommitted activity for optimistic LPs. Fast GVT also can improve caching behavior, since it can help keep the working set small.

Here, we consider three major variants for GVT computation: (1) whenever the engine initiates a GVT computation, it blocks until a new GVT is computed, (2a) GVT computation and engine's main loop can be concurrently active, with the same two-sided, inter-processor communication being used for both event exchange and GVT messages, and (2b) just as in 2a, GVT and event loops are concurrent, but they are independent with respect to communication, with the GVT computed using a direct memory access mode via one-sided communication. These three variants are described next.

### A. Synchronous Two-sided GVT

Similar to the synchronous variant of the *lower bound time stamp* (LBTS) algorithm (over reliable transport) in Perumalla and Fujimoto [7], and similar to the synchronous algorithm given in Holder and Carothers [10], Algorithm 1 is a generalization of [7], [11], but enhanced to support conservative as well as optimistic execution with lookahead. The conceptually simple approach in Algorithm 1 assumes synchronous execution and two-sided communication. It repeatedly computes global summation of the count of events sent and received by each processor, thereby indicating the presence of "transient" messages floating in the network. When all processors detect the absence of any transient messages in the system, they proceed to compute the global minimum of their local virtual times, thus giving global virtual time. In optimistic discrete event executions, retractions (anti-messages) are treated as regular events by using their timestamps just as those for regular messages.

Unlike previous works [10], Algorithm 1 takes into account the lookahead value (which could be zero if needed by the model), and hence is usable by both conservative and optimistic executions equally well.

**Algorithm 1** Synchronous GVT algorithm invoked from within the main loop whenever a new GVT is needed

1: $nsent \leftarrow$ no. of events sent so far to other processors
2: $LVT \leftarrow \min($ minimum of all local timestamps, $\infty)$
3: $LA \leftarrow$ lookahead from this to any other processor
4: $nrecd \leftarrow 0$
5: **repeat**
6: $\quad \delta \leftarrow$ blocking reduction $\sum(nsent - nrecd)$
7: $\quad$ **if** $\delta = 0$ **then**
8: $\quad\quad GVT \leftarrow$ blocking reduction $\min(LVT + LA)$
9: $\quad$ **else**
10: $\quad\quad$ **while** any event $E$ from a processor is available **do**
11: $\quad\quad\quad$ Receive $E$
12: $\quad\quad\quad$ $nrecd$++
13: $\quad\quad\quad$ $LVT \leftarrow \min(LVT, E.timestamp)$
14: $\quad\quad$ **end while**
15: $\quad$ **end if**
16: **until** $\delta = 0$
17: $nsent \leftarrow 0$

### B. Asynchronous Two-sided GVT

Moving to an asynchronous formulation of the GVT computation requires rewriting the main discrete event execution loop as one in which GVT-related messaging and computation is interleaved with local event processing. This is shown in Algorithm 2. A variable $d$ is used as a counter of the number of GVT computations performed so far, also termed as the epoch number. Each computation proceeds as sequence of *trials*, which are successive reductions to determine the number $\delta$ of transient events "in flight," computed as the reduction with the addition operator on the difference between the number of events sent in previous epoch and the number received in previous or current epoch. Together with the summation, a reduction for the global minimum is performed on the minimum local timestamps at each processor (line 13). When $\delta$ becomes zero, clearly, the globally reduced minimum time is usable as a (non-decreasing) GVT value (line 20). If $\delta$ is non-zero, then, another asynchronous reduction must be started to determine if there has been progress in event delivery (line 24).

### C. Asynchronous One-sided GVT

The one-sided GVT operates exactly as in Algorithm 2, with one major difference. Non-blocking GVT must perform its synchronization via messaging, which gets multiplexed along with incoming and outgoing event communication. Since GVT messages compete with event messages, the mixed communication can impose latency for GVT messages, thereby delaying GVT completion. One-sided communication, on the other hand, can be used to exchange GVT messages with minimal delay which is independent of the event communication.

The memory organization on each processor is shown in Figure 1, with arrows showing the potential one-sided transfer of data from the send buffers of processor $P_i$ to the receive buffers of processor $P_j$. Since GVT computation proceeds asynchronously with the main event processing loop, some processors complete a given GVT epoch $d$ earlier than others and may proceed to initiate the next epoch $d+1$. Analogously, a trial $r$ within an epoch $d$ may complete on one processor which proceeds to its next trial $r+1$, thereby sending information belonging to epoch $d$ and trial $r+1$ while the receiving processor may still be in the process of completing the earlier epoch $d$, trial $r$. Hence, at any given moment, every processor must maintain four different blocks of receivable data: $\{(d,r),(d,r+1),(d+1,r),(d+1,r+1)\}$, to keep the asynchronous computations independent of each other.

Using a tree topology optimized for hierarchical (application-level, asynchronous) reductions, the inter-processor structure is fixed for GVT messaging, determined and initialized before beginning the main simulation loop. The unit of memory layout for the GVT data structures is a fixed message size (a `C struct`) defined to hold a GVT message type. Additionally, room for *jumpstart* messages is also allocated such that processors may jumpstart other processors (within or outside its hierarchy) to begin participating in a GVT computation. Some processors may need to be informed so, because, during their own asynchronous event processing, they may not themselves need any additional GVT advances until they run out of local event execution work. The jumpstart messages thus help inform processors when they need to participate in GVT computations started by other processors.

**Algorithm 2** Asynchronous GVT algorithm within main execution loop, for both two-sided and one-sided communication

1: $d$: GVT epoch number, initially 0
2: $r$: Trial number within a GVT epoch
3: $nsent_d$: no. of events sent in epoch $d$
4: $nrecd_d$: events sent in epoch $d$ received in $d$ or $d+1$
5: $LVT_d$ : $\min($ all local timestamps in epoch $d$ $,\infty)$
6: $LA$: lookahead from this to any other processor
7: $isactive$: flag that GVT is being computed (initially false)
8: **loop**
9:   **if** (a new GVT is needed) **and** (**not** $isactive$) **then**
10:     $d++$
11:     $r \leftarrow 0$
12:     $isactive$ ←true
13:     Start asynchronous reduction of all processors' $\sum(nsent_{d-1} - nrecd_d)$ and $\min(LVT_d + LA)$
14:   **end if**
15:   **if** $isactive$ **then**
16:     Advance the active asynchronous reduction
17:     **if** asynchronous reduction completed **then**
18:       $\delta \leftarrow reduced \sum(nsent_{d-1} - nrecd_d)$
19:       **if** $\delta = 0$ **then**
20:         $GVT$ ←reduced $\min(LVT_d + LA)$ value
21:         $isactive$ ←false
22:       **else**
23:         $r++$
24:         Start asynchronous reduction of all processors' $\sum(nsent_{d-1} - nrecd_d)$ and $\min(LVT_d + LA)$
25:       **end if**
26:     **end if**
27:   **end if**
28:   Perform local event processing {conservative or optimistic}
29:   **for** each event $E_s$ being sent to another processor **do**
30:     $nsent_d++$
31:     Tag $E_s$ as sent in epoch $d$
32:   **end for**
33:   **while** any event $E_r$ from a processor is available **do**
34:     Receive $E_r$ and its tag $d_r$
35:     $nrecd_{d_r}++$
36:     $LVT_d \leftarrow \min(LVT_d, E_r.timestamp)$
37:   **end while**
38:   ...
39: **end loop**

The potential advantages of one-sided messaging are: (1) GVT messaging becomes separated from event communication, thereby eliminating competition, and its resultant latency increase, for GVT messages, (2) overheads of dynamic memory remapping is avoided due to static inter-processor

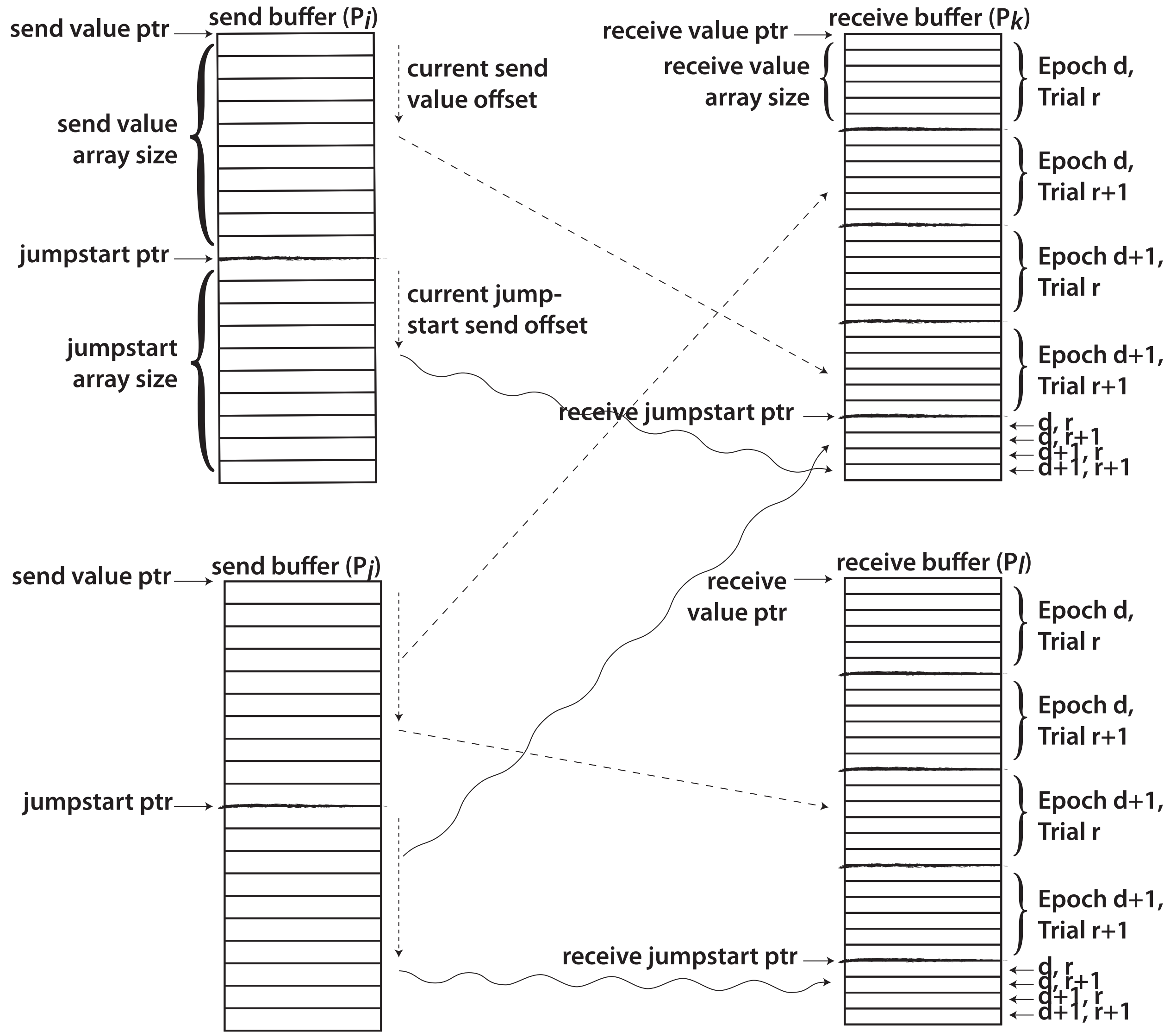



Fig. 1. Data structures for one-sided communication-based GVT. Each in the send or receive buffers includes the $(LVT + LA, nsent - nrecd)$ values of the corresponding sender. Dashed lines represent `put` operations of reduced values, while the squiggly lines represent `put` operations of *jump start* messages to initiate reduction on the receiver side.

messaging structure for GVT messages.

We now present implementation details of the algorithms incorporated into the $\mu sik$ discrete event execution engine.

## III. Implementation

The implementation and experimentation were performed on a Cray XT5 system in which each node consists of 2 hex-core AMD Opteron 2435 (Istanbul) 2.6GHz processors and 16GB of memory. The nodes are connected through Cray's SeaStar 2+ 3D torus interconnect.

All inter-processor event communication is performed using traditional two-sided communication via MPI. The GVT message exchange is also performed using MPI for the two-sided GVT algorithms, and via the Portals [12] one-sided interface for one-sided communication. Asynchrony with respect to event messaging is realized via `MPI_Iprobe()` for two-sided communication, and via `PTL_EQGet()` for one-sided notifications. The synchronous two-sided GVT algorithm uses `MPI_Allreduce()` for the blocking reduction of the transient message counts and local virtual time values. It is also easy to implement because complexities of multiple concurrent epochs are absent due to the fact that all processors are always at the same epoch number and the same trial number. The asynchronous two-sided GVT algorithm is implemented with user-level reductions performed via an optimized butterfly pattern, using MPI messaging for exchanging the reduction messages. Communication for asynchronous one-sided GVT messaging is implemented using the Portals interface.

The Portals API on the Cray XT5 is implemented using Portals Network Access Layer (NAL). The Portals NAL provides a bridge between the Portals API and the SeaStar Network Interface Card (NIC) and utilizes Basic End-to-

End Reliability (BEER) protocol for ensuring reliability and performing credit-based flow control.

In the one-sided GVT algorithm implementation, Portals is initialized with memory descriptors (MDs) used for `put` operations configured with infinite threshold, and bound using `PtlMDBind()` with the `PTL_RETAIN` setting to make MDs reusable for later sends. To send, `PtlPutRegion()` is used with `PTL_NOACK_REQ` since acknowledgments for send completions are not needed by our GVT algorithm. For notification of completion of one-sided put operations for GVT messages, we subscribe to the `PTL_EVENT_SEND_END` notification. Similarly, `PTL_EVENT_PUT_END` notification is subscribed to for notification of incoming GVT messages. All destination memory locations of all one-sided puts are managed on the sender-side, and hence, `PTL_MD_MANAGE_REMOTE` is used on all sender-side MDs. The `PTL_EQGet()` and `PTL_EQWait()` calls are used to process all Portals notifications asynchronously. Since the maximum number of outstanding puts are bounded per GVT (epoch), it is possible to select a Portal event queue size such that no notifications would be dropped, and hence `PTL_EQ_DROPPED` would be flagged as an error condition.

The implementation of both non-blocking and one-sided GVT algorithms is carefully done to ensure that *no barriers are ever invoked from the main loop.*

The MPI option `MPICH_PTL_MATCH_OFF` was used to make MPI perform message matching for the underlying Portals device. In synchronous two-sided operation, we have found that this provides a noticeable performance improvement due to the latency-sensitive nature of PDES applications.

All of the benchmarks that are used to evaluate the GVT algorithm performance are written as applications using the same simulation engine, $\mu sik$, which is one of only two PDES engines reported to date to scale to over $10^5$ processor cores. All the GVT algorithms have been implemented into $\mu sik$, any one of which can be chosen by the user at runtime initialization via an environment variable specification.

## IV. Performance Analysis

We examine the dynamics of discrete event execution exercised with the major GVT algorithms presented in this work scaled up to 129,024 processor cores. In order to evaluate the efficacy of each GVT algorithm, we selected four PDES benchmarks which represent a wide cross-section of PDES application characteristics, from varied event densities to mixed messaging and event computation intensities.

All of the software used in this performance study was compiled with the Portland Group (pgi) compiler version 2.2.73 with `-O3 -fast` compilation flags.

### A. Execution Benchmarks

We use the following four PDES applications that run over $\mu sik$, thus automatically inheriting the runtime benefits of all the three GVT algorithm implementations and their optimizations incorporated into $\mu sik$.

*1) RCPHOLD:* The PHOLD application [13] is a de facto PDES benchmark used to exercise the underlying simulator's efficiency in event processing, message transmission and reception to destination LPs and, if applicable, rollback efficiency. PHOLD is a synthetic benchmark with little event computation other than random number generation to determine the virtual time increments and destination LPs. PHOLD can be executed conservative mode as well as optimistic mode.

PHOLD can be configured to send to random or a subset of destinations. We define a value, *neighbor reach*, such that a processor only sends to remote processors whose identifiers are within a $\pm$ neighborhood of its own. Events can also be sent to self. Outgoing events are timestamped with a exponentially distributed timestamp with a mean of 1.0 plus lookahead.

The PHOLD benchmark can be configured into specific structures affecting event density and messaging behavior, two of which are used for evaluation. For the present purposes, we denote *structure* as a tuple of $(\sigma, \gamma)$, where $\sigma$ is the number of LPs per core, and $\gamma$ is a specific parameter for the simulation. For PHOLD, $\gamma$ is the multiplier for the message population of the simulation. Thus, $\sigma \times \gamma \times \omega$ gives the total message population of the entire simulation across $\omega$ cores.

The "RC" moniker of RCPHOLD stands for reverse computation. Instead of storing the state of the simulation prior to each event processed to facilitate rollback in optimistic simulations. When a rollback occurs, the simulator performs a sequence of undo operations that restore the state of the simulation to the proper good state before incorrect events were executed. This is a classic space-time tradeoff where, to rollback the simulation, significant memory savings may be obtained in exchange for some computational overhead.

*2) RCREDIF:* Another significant PDES benchmark used is called RCREDIF [14], which is a large-scale epidemiological outbreak simulation based on a reaction-diffusion model. It uses a novel discrete event formulation of the phenomenon, and a new reverse computation-based model as rollback support in its scalable optimistic simulation. Organized in terms of a number of individuals per location ($\gamma$), a number of locations per region ($\sigma$), and a region per processor, RCREDIF simulates probabilistic transition state machines at the level of each individual within populations Similar to RCPHOLD, RCREDIF also can be executed both in conservative mode as well as optimistic mode using reverse computation, and also employs the *neighbor reach* specification (similar to RCPHOLD) in determining the remote processors selected as potential destinations.

Due to the amount of computation involved in reversing an event, the rollback cost per event is relatively high in RCREDIF. Thus, even if the rollback *length* is small in an RCREDIF simulation run, the total rollback runtime *overhead* can be relatively high.

*3) $\mu\pi$:* $\mu\pi$ [15] is a software-based experimentation platform for testing synthetic and real unmodified MPI programs. $\mu\pi$ multiplexes virtual MPI ranks per real rank (the ratio to be referred to as *LPX*) for execution over simulated virtual

platforms through $\mu sik$'s process-oriented PDES framework.

*a) Barrier Test:* The barrier test benchmark aims to stress-test multiple items of interest: (a) ability to instantiate and advance millions of virtual ranks on the simulation time axis, (b) performance under very tight coupling among ranks, especially with regard to stringent characteristics of their virtual interconnection network, and (c) ability for a high level of multiplexing for maximum efficiency. In the benchmark, every rank repeatedly joins a barrier by invoking `MPI_Barrier()`, and querying the time taken by each barrier via the times returned by `MPI_Wtime()`. Also between each pair of barriers, each rank advances simulation time by one millisecond to model a relatively coarse-grained computation.

*b) Ping Test:* The ping test benchmark is used to measure bandwidth and latency between pairs of communicating MPI ranks. This ping test has virtual ranks arranged in a naturally-ordered ring topology. The sender sends data to the next higher virtual rank while receiving data from the lower virtual rank. If the virtual rank number is even, it performs a blocking send followed by a blocking receive. The order of operations is reversed for odd-numbered virtual ranks. These operations are timed via calls to `MPI_Wtime()` for bandwidth and latency measurement.

These operations are iterated successively from 8 bytes to the maximum specified test message size, where the length of each message is doubled for each trial until the maximum limit is reached. Note that for testing, the $\mu sik$ messaging layer implemented a constant payload length of approximately 50KB even if the message length was smaller than this amount.

### B. Experiment Setup

Each of the GVT algorithms outlined was tested within each application. For labels in all of the following charts, we use a 3-tuple `(X Y Z)`. `X` is the synchronization strategy employed: either `C` for conservative or `O` for optimistic. `Y` denotes usage of a one-sided GVT algorithm (utilizing the Portals interface within $\mu sik$ time management) where `T` notes that the feature was enabled and `F` if it was disabled. `Z` signifies whether or not the synchronous GVT algorithm was used. Thus, `(. F T)` refers to the synchronous two-sided GVT algorithm, `(. F F)` to the asynchronous two-sided GVT algorithm and `(. T F)` to the one-sided asynchronous GVT algorithm.

Combinations of lookahead, structure, synchronization strategy and GVT algorithms were varied for each benchmark. Lookaheads were varied across the RCPHOLD and RCREDIF benchmarks, ranging from very low to very high values of lookahead. Additionally, the structure ($\sigma$, $\gamma$) of each application was varied between `(10,1000)` and `(100,100)`. Thus the message population remained constant between structures per core, but the number of LPs per core varied.

For $\mu\pi$ benchmarks, lookahead was fixed based on the network properties. Here we selected a prototypical fast (i.e., latency of 10$\mu$s and bandwidth of 1Gb/s) and very fast (i.e., latency of 1$\mu$s and bandwidth of 10Gb/s) network specification to determine the lookahead. The "structure" of the $\mu\pi$ benchmarks is simply *LPX* (i.e., number of virtual MPI ranks multiplexed on each real rank), where values of 128 and 1024 were chosen to showcase light and very heavy multiplexing.

TABLE I
SYMBOLS USED IN CHARTS

| Symbol | Description |
|---|---|
| $\varepsilon$ | Aggregate committed event rate (millions events/sec) |
| $\lambda$ | Number of GVT epochs |
| $\rho$ | Maximum number of rollbacks observed on a single core |
| $\alpha$ | Factor of improvement of asynchronous algorithms over synchronous two-sided GVT algorithm |
| `F F` $\alpha$ | Factor of improvement of asynchronous two-sided GVT algorithm over synchronous two-sided GVT algorithm |
| `T F` $\alpha$ | Factor of improvement of asynchronous one-sided GVT algorithm over synchronous two-sided GVT algorithm |

TABLE II
NOTATIONS USED IN CHARTS

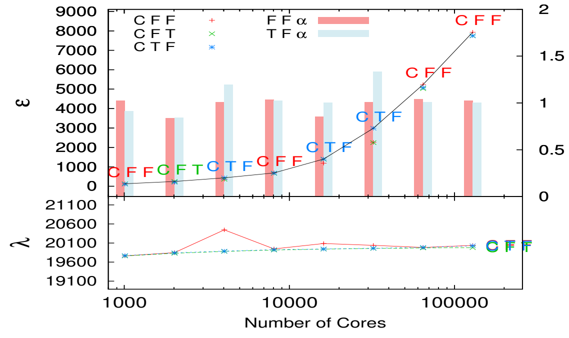
| Notation | Description |
|---|---|
| $(\sigma, \gamma)$ | Structure of simulation |
| *LPX* | Number of virtual MPI ranks multiplexed on each real MPI rank |
| `(C . .)` | Conservatively synchronized simulation |
| `(O . .)` | Optimistically synchronized simulation |
| `(. F T)` | Synchronous two-sided GVT algorithm |
| `(. F F)` | Asynchronous two-sided GVT algorithm |
| `(. T F)` | Asynchronous one-sided GVT algorithm |

The simulation end times were set to 1000 simulated seconds and 168 simulated hours for all scenarios in RCPHOLD and RCREDIF, respectively. $\mu\pi$ barrier test simulated one virtual barrier for all *LPX*, while $\mu\pi$ ping test simulated up to 1KiB and 64KiB of data transfer in the *LPX*=1024 and *LPX*=128 structures, respectively.

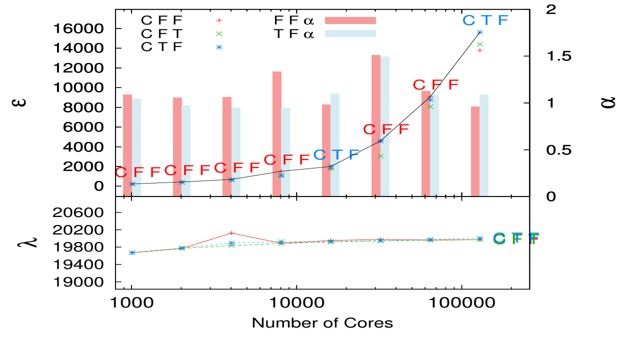
For the following charts, $\varepsilon$ denotes the aggregate committed event rate in millions of events/sec which is plotted on the primary ordinate. Each individual data point for the three GVT algorithms tested is plotted while the best performing GVT algorithm (i.e., the algorithm achieving the highest $\varepsilon$) is noted at each core count. A line joining the maxima is drawn through each of these best-performing numbers to visually show a trendline of performance as the simulation is scaled. Additionally, charts include $\alpha$ bars which denote the factor of improvement over `(. F T)` or the synchronous two-sided GVT algorithm for the asynchronous GVT algorithms *i.e.*, `(. F F)` and `(. T F)` on the secondary ordinate. Secondary plots on the following charts may include $\lambda$, which denotes the number of GVT epochs or $\rho$, which denotes the maximum number of rollbacks occurring on a single core within the entire simulation for selected optimistic executions. The symbols and notations are summarized in Table I and Table II.

### C. RCPHOLD Results

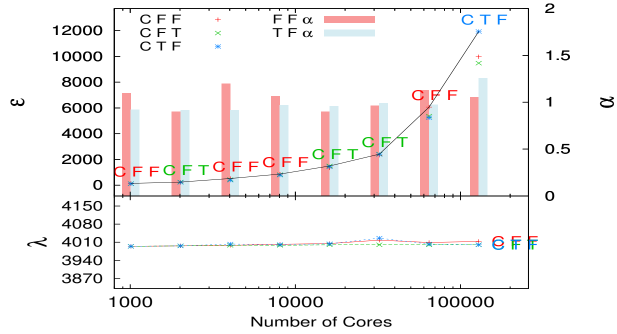
For conservatively synchronized RCPHOLD benchmarks at low lookahead of 0.1 shown in Figure 2a and Figure 2b, we



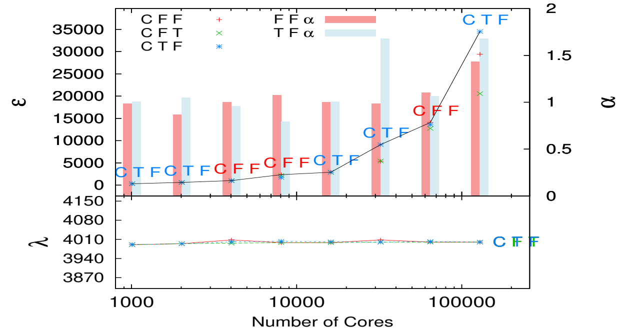

(a) LA=0.1, (10,1000)




(b) LA=0.1, (100,100)

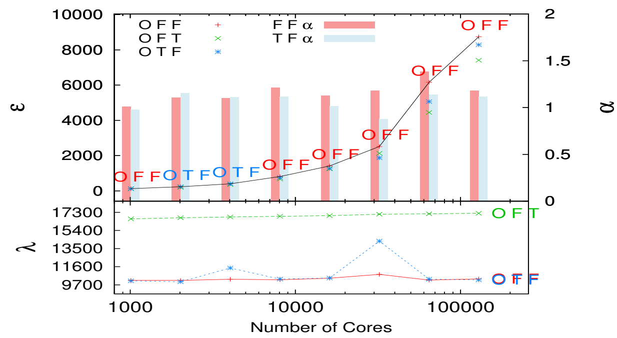



(c) LA=0.5, (10,1000)

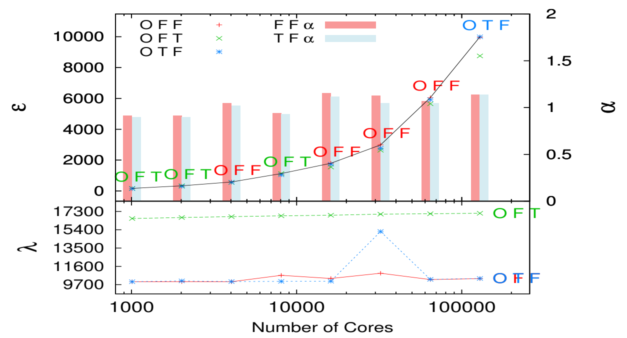



(d) LA=0.5, (100,100)

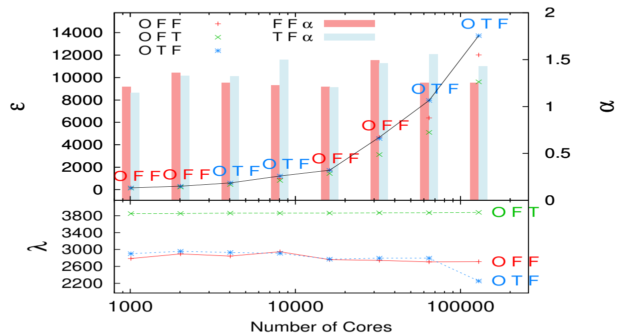
Fig. 2. RCPHOLD Conservative Synchronization



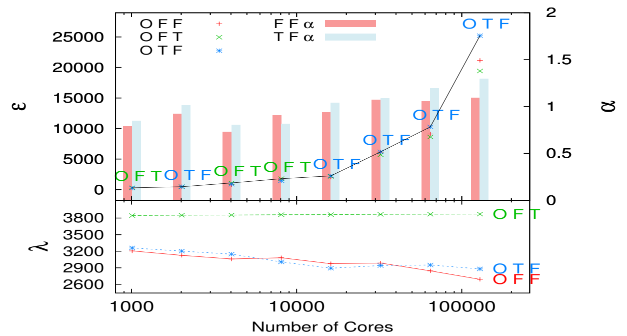

(a) LA=0.1, (10,1000)




(b) LA=0.1, (100,100)




(c) LA=0.5, (10,1000)




(d) LA=0.5, (100,100)

Fig. 3. RCPHOLD Optimistic Synchronization

observe nearly in all cases that asynchronous GVT algorithms performed no worse than synchronous two-sided GVT, and sometimes provided significant improvements in $\varepsilon$ – up to $1.5\times$ in the structure of `(100,100)`. Interestingly, $\lambda$ remained nearly the same for all GVT algorithms and holds through all tested core counts. Thus, the runtime difference and performance improvement shown by both asynchronous GVT algorithms is from decrease in wall-clock time consumed by these algorithms per GVT computation.

At high lookahead of 0.5 shown in Figure 2c and Figure 2d, we observe similar performance from both synchronous and asynchronous GVT algorithms at smaller scales. As the simulation is scaled to 32K processor cores and beyond, asynchronous GVT algorithms tend to cope with larger number of processor cores better as $\varepsilon$ improvements exceeding $1.5\times$ is observed in the `(100,100)` case. We can reason here that the increased amount of lookahead lowers the total amount of synchronization burden across the entire simulation which becomes increasingly more taxing as the simulation is spread across more processor cores. A $5\times$ decrease is observed in $\lambda$, which is inversely correlated with $5\times$ the increase in the amount of lookahead, as expected.

Optimistically synchronized RCPHOLD provides further insight into how the speed, behavior and quality of information delivered by the underlying GVT algorithms can drastically impact the performance of this particular PDES benchmark.

In all cases of lookahead as shown in Figure 3, we see that in nearly all cases, both asynchronous GVT algorithms provide at least the performance of synchronous two-sided GVT algorithm but can often accelerate the simulation much faster showing consistent $1.2\times$ to over $1.5\times$ the performance of synchronous two-sided GVT, especially in the `(10,1000)` structure cases. The striking detail that comes forth through all of the RCPHOLD optimistic charts is the significant difference in $\lambda$ for the synchronous two-sided GVT algorithm. Frequent GVT computation is not necessarily a detriment to overall performance. In fact, having fresh GVT information can reduce the number of potential incorrect events processed (and thus the number of rollbacks) in an optimistic parallel simulation. However, this generalization only holds if the cost of the GVT computation is relatively inexpensive compared to the cost of rollback. Since RCPHOLD is a synthetic benchmark that is not computationally intense, rollback costs are very inexpensive. *Thus, for RCPHOLD, the rollback cost is significantly less than GVT computation cost.*

We can clearly observe these dynamics in RCPHOLD. $\lambda$ is more than $1.5\times$ in the synchronous two-sided GVT cases over both the asynchronous cases, yet there are no rollbacks in the synchronous two-sided GVT cases while there are rollbacks present in the both asynchronous cases (charts for rollback data were omitted due to space considerations). We see that in certain cases, such as shown in `(10,1000)` structure in Figure 3a and Figure 3c, at larger core counts, the $\varepsilon$ gap widens as the cost per $\lambda$ increases with the number cores. It is clear here that the quality of the GVT information delivered by both asynchronous GVT algorithm is no less than, if not better than, that of the synchronous two-sided GVT algorithm, yet incurs less overhead by way of smaller $\lambda$.

Figure 6a shows the overall speedup trends for the best and worst committed event rate trends. Speedup is measured over the base of 1008 cores for their respective GVT algorithm (i.e., self-relative speedup). The best and worst observed $\varepsilon$ for RCPHOLD are both using conservative synchronization at lookahead of 0.5 with a structure of `(100, 100)` and lookahead of 0.1 with a structure of `(10, 1000)`, respectively.

### D. RCREDIF Results

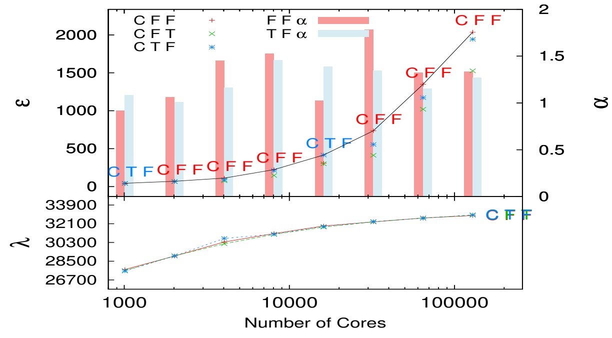
RCREDIF with very low lookahead of 0.01 is shown in Figure 4a and Figure 4b. Similar to RCPHOLD, we observe that both asynchronous GVT algorithms provide better performance than the synchronous two-sided GVT algorithm at all core counts. $\lambda$ remains nearly identical for all GVT algorithms as the RCREDIF application is scaled out. Clearly, the cost per $\lambda$ is the differentiating determinant for $\varepsilon$ and thus, $\alpha$.

Due to space considerations the RCREDIF charts for low lookahead of 0.1 and high lookahead of 0.5 were omitted, but generally followed the same trends as very low (0.01) or very high (1) lookahead, respectively.

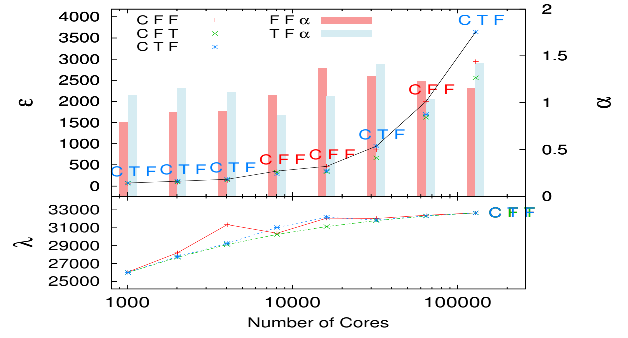
In the cases of higher lookahead of 1 in Figure 4c and Figure 4d, the difference between the GVT algorithms is lessened due to the relatively small $\lambda$ during the simulation execution. Even in most of these cases, we observe that both asynchronous GVT algorithms provide $\varepsilon$ performance on par to that of the synchronous two-sided GVT algorithm, if not better. This clearly becomes the case at 129,024 cores where the use of the asynchronous GVT algorithms, and one-sided in particular, provide significant performance gains.

In contrast to the RCPHOLD synthetic benchmark, RCREDIF is a real application that has significant event computation costs. Thus, rollbacks are expensive in comparison to those found in RCPHOLD. At very low lookahead of 0.01 under optimistic synchronization for RCREDIF shown in Figure 5a and Figure 5b, we observe that $\rho$ for the synchronous two-sided GVT algorithm is significant while both asynchronous GVT algorithms incur zero rollback. This inversely correlates with $\varepsilon$ where we observe $1.2\times$ to over $2\times$ $\alpha$. In these very low lookahead scenarios, $\lambda$ tends to be very large (i.e., 16K to 30K+ computations) over the course of the execution. The speed and frequency of the GVT algorithm comes in to play for these small lookaheads. Both asynchronous GVT algorithms exhibit larger $\lambda$, providing more up-to-date GVT information without sacrificing simulation speed of event computation. The synchronous two-sided GVT algorithm on the other hand synchronizes less frequently, up to nearly 50% less, yet performs significantly worse at scale. The speed of the GVT algorithm clearly impacts the event execution dynamics: as potentially more incorrect events are executed they must ultimately be rolled back, incurring significant synchronization overhead cost.

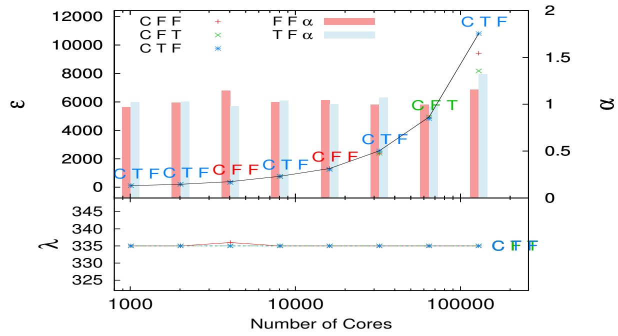
For the very high lookahead of 1 as shown in Figure 5, the trend reverses, where the synchronous two-sided GVT algorithm incurs mostly no rollbacks while both asynchronous



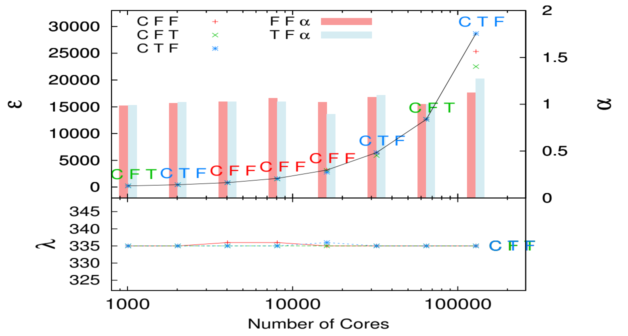
(a) LA=0.01, (10,1000)



(b) LA=0.01, (100,100)

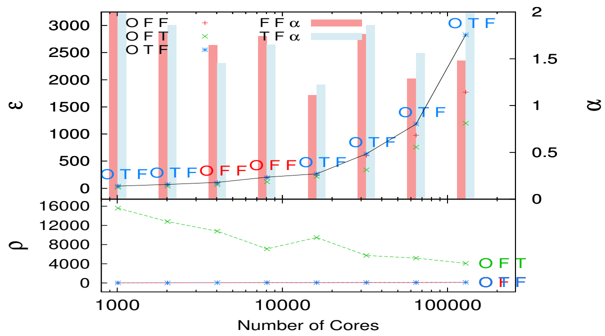


(c) LA=1, (10,1000)

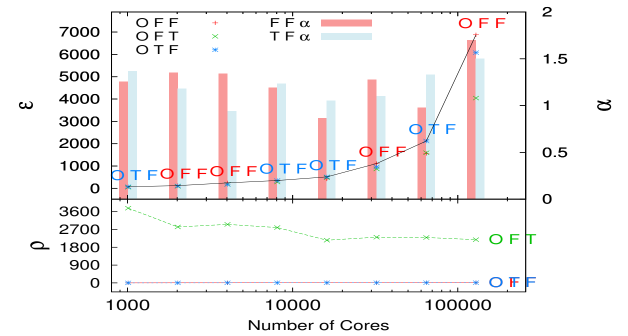


(d) LA=1, (100,100)

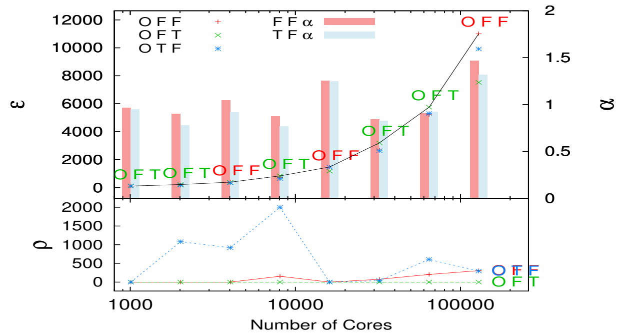
Fig. 4. RCREDIF Conservative Synchronization



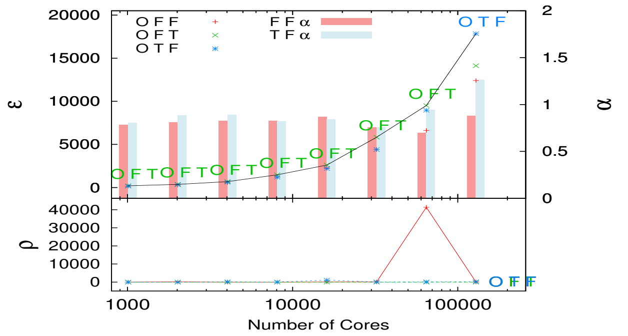
(a) LA=0.01, (10,1000)



(b) LA=0.01, (100,100)

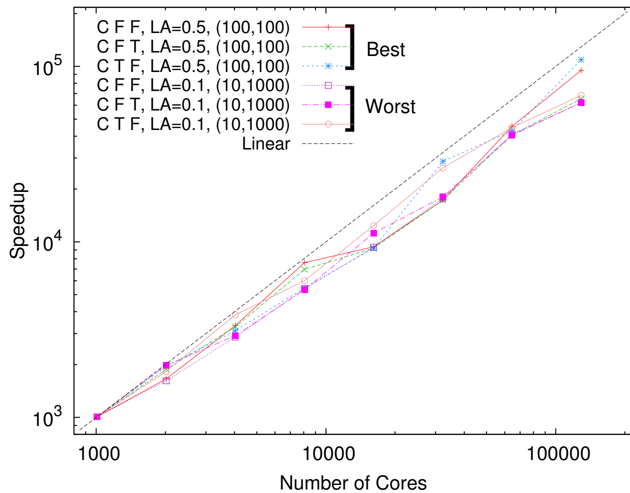


(c) LA=1, (10,1000)

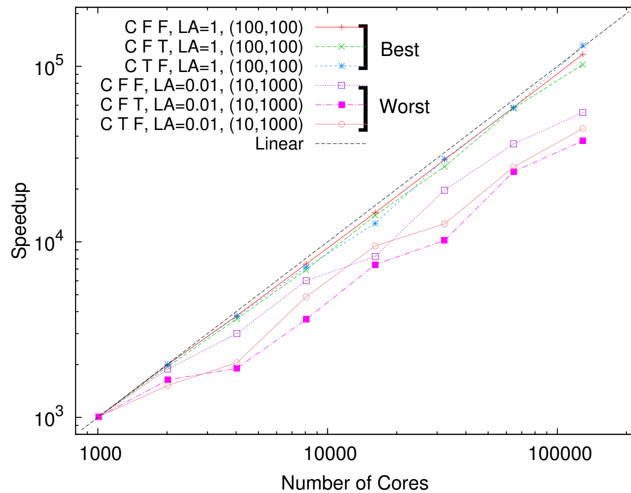


(d) LA=1, (100,100)

Fig. 5. RCREDIF Optimistic Synchronization

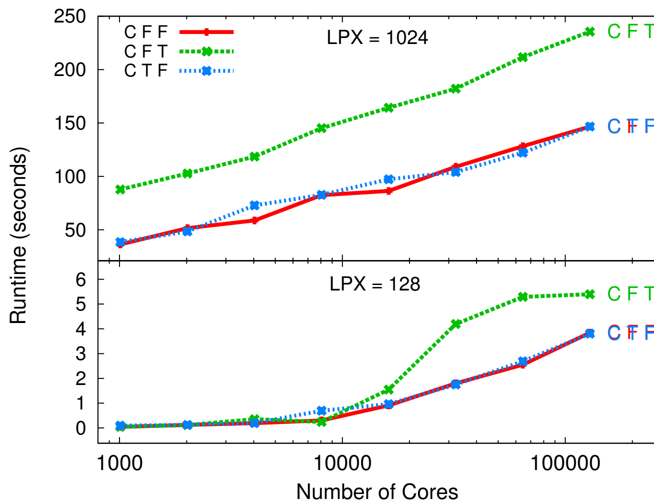



(a) RCPHOLD Self-relative Speedup

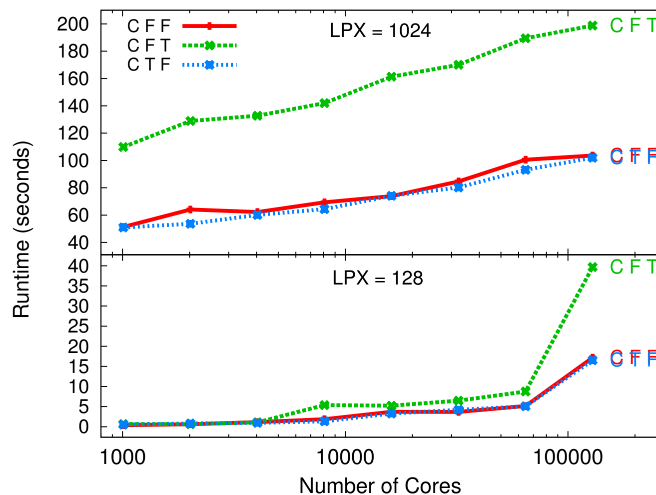



(b) RCREDIF Self-relative Speedup

Fig. 6. Self-relative Speedups for Best and Worst Committed Event Rate Trends




(a) Barrier Test




(b) Ping Test

Fig. 7. $\mu\pi$ Runtime Performance

GVT algorithms do. Since $\lambda$ is reduced significantly in these cases due to high lookahead, each GVT advance is important to ensuring minimal amount of rollbacks. The scenarios employing the synchronous two-sided GVT algorithm synchronize more frequently: up to 50% more than both asynchronous cases. However, since $\lambda$ in these runs are relatively small compared to the total elapsed runtime of the simulation, the additional number of $\lambda$ do not significantly interfere with event computations. This tends to prevent excessive rollbacks and thus lower $\varepsilon$ performance as shown in the respective charts. At 129,024 processors, the asynchronous GVT algorithms outperform the synchronous two-sided GVT algorithm. With $\lambda$ and $\rho$ metrics remaining consistent with the prior data point at 64,512 cores, the drop-off in performance for synchronous two-sided GVT performance might be attributed to the increased wallclock time incurred per $\lambda$. Further experimentation is needed to verify the cause of the performance degradation for synchronous two-sided GVT at very large-scale for RCREDIF.

Figure 6b shows speedup for the best and worst case committed event rates for RCREDIF. The best and worst observed $\varepsilon$ for RCREDIF are both using conservative synchronization at lookahead of 1 with a structure of `(100, 100)` and lookahead of 0.01 with a structure of `(10, 1000)`, respectively.

### E. $\mu\pi$ Results

In the $\mu\pi$ benchmarks as shown in Figure 7, the performance difference between synchronous two-sided GVT algorithms and asynchronous GVT algorithms are clearly pronounced. Here, $\lambda$ is approximately equal for all scenarios, thus indicating that the time to complete GVT computations

in the synchronous two-sided GVT algorithm is significantly longer than both asynchronous GVT cases. We observe a 1.6× performance improvement in runtime for the asynchronous one-sided GVT algorithm over the synchronous two-sided GVT algorithm at 129,024 processor cores simulating over 132 million virtual MPI ranks in the *LPX*=1024 case for barrier test as shown in Figure 7a. Similarly for the ping test shown in Figure 7b, there is a 1.95× improvement in runtime for the asynchronous one-sided GVT algorithm over the synchronous two-sided GVT algorithm at the same scale for *LPX*=1024. For the lightly multiplexed cases of *LPX*=128, the runtime performance differential between synchronous and asynchronous GVT algorithms begins to appear at scale when the number of cores exceeds approximately 8K.

The difference in performance between GVT algorithms can be attributed to the time-slicing nature of process-oriented PDES where multiple virtual threads are multiplexed on top of a single real execution thread of the main loop. As the number of virtual contexts are increased, the proportional amount of time taken by the GVT algorithm becomes larger in relation to the amount of time given per context switch to each virtual thread. Thus, the effects of a slower, synchronous GVT algorithm becomes apparent on high multiplexing counts.

## V. Summary

The performance data gathered in this study to the scale of tens of thousands of processors offers confirmation of some general knowledge in PDES, but also uncovers new insight into discrete event dynamics at scale.

1) There exists a cost trade-off between GVT frequency and rollbacks. Executions which incur higher per-event rollback costs can benefit from more frequent GVT computations. GVT algorithms that complete faster, such as in the asynchronous approaches, can lead to significant performance gains by minimizing rollbacks at the relatively smaller expense of more frequent GVT computations.
2) Cost per rollback is not constant as executions scale. RCREDIF shows that, as an execution is scaled out, the gap between GVT algorithms which lead to little or no rollback provides significant gains in simulation performance.
3) Asynchronous GVT algorithms tend to almost always perform at least as well as their synchronous counterpart. In the majority of cases, the asynchronous GVT algorithms accelerate the execution by spending less time in synchronization overheads. The notable exception to this rule, as we have observed, comes in optimistic execution at high lookahead where the synchronous two-sided GVT algorithm tends to synchronize more frequently, thus, in effect, preventing possibility of staggered execution and rolled-back event computation. However, this exception only seems to be limited to executions with less than $10^5$ cores.
4) Process-oriented PDES which time-multiplex multiple contexts on to a single core can benefit from asynchronous GVT algorithms with greater multiplexing levels and/or at larger core counts. As the amount of processor time per thread becomes more scarce at higher multiplexing counts, the relative amount of time spent in GVT rises. Thus, the asynchronous nature of GVT algorithm is beneficial in allowing events to be processed asynchronously with GVT computation.

Overall, synchronous and asynchronous GVT computation on large systems achieve performance with high efficiency in discrete event execution. While additional analysis is possible, such as a study of the influence of the frequency of global synchronization on optimistic execution, the high event rates (of several billions of events executed per wall clock second) are very encouraging for a broad class of PDES applications.

## References


[1] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[2] K. S. Perumalla, "Switching to high gear: Opportunities for grand-scale real-time parallel simulations," in *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2009, pp. 3–10.

[3] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *J. Parallel Distrib. Comput.*, vol. 18, pp. 423–434, August 1993.

[4] J. Reynolds, Paul, C. Pancerella, and S. Srinivasan, "Design and performance analysis of hardware support for parallel simulations," *Journal of Parallel and Distributed Computing*, vol. 18, no. 4, pp. 435–453, 1993.

[5] M. Rosu, K. Schwan, and R. M. Fujimoto, "Supporting parallel applications on clusters of workstations: The intelligent network interface approach," in *IEEE Symposium on High Performance Distributed Computing*, 1997.

[6] D. Chen and B. K. Szymanski, "Dsim: Scaling time warp to 1,033 processors," in *Winter Simulation Conference*. Orlando, FL: IEEE, 2005.

[7] K. Perumalla and R. Fujimoto, "Virtual time synchronization over unreliable network transport," in *Proceedings of the 15th workshop on Parallel and distributed simulation*. IEEE Computer Society, 2001, pp. 129–136.

[8] C. D. Carothers and K. S. Perumalla, "On deciding between conservative and optimistic approaches on massively parallel platforms," in *Winter Simulation Conference*. IEEE Computer Society, 2010, pp. 678–687.

[9] K. S. Perumalla, "$\mu sik$: A micro-kernel for parallel/distributed simulation systems," in *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 2005, pp. 59–68.

[10] A. O. Holder and C. D. Carothers, "Analysis of time warp on a 32,768 processor ibm blue gene/l supercomputer," in *2008 Proceedings European Modeling and Simulation Symposium (EMSS)*, August 2008.

[11] D. W. Bauer Jr., C. D. Carothers, and A. Holder, "Scalable time warp on blue gene supercomputers," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 2009, pp. 35–44.

[12] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. Underwood, "Implementation and performance of portals 3.3 on the cray xt3," in *Cluster Computing, 2005. IEEE International*, 2005, pp. 1–10.

[13] K. S. Perumalla, "Scaling time warp-based discrete event execution to $10^4$ processors on the blue gene supercomputer," in *International Conference on Computing Frontiers*, Ischia, Italy, 2007, pp. 69–76.

[14] K. S. Perumalla and S. K. Seal, "Discrete event modeling and massively parallel execution of epidemic outbreak phenomena," *Transactions of the Society for Modeling and Simulation International*, vol. in print, p. doi:10.1177/0037549711413001, 2011.

[15] K. S. Perumalla, "$\mu\pi$: A scalable and transparent system for simulating mpi programs," in *Proceedings of the 3rd International Conference on SIMUTools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.