Discrete Event Execution with One-Sided and Two-Sided GVT Algorithms on 216,000 Processor Cores

KALYAN S. PERUMALLA, Oak Ridge National Laboratory
ALFRED J. PARK, Microsoft Corporation
VINOD TIPPARAJU, Advanced Micro Devices, Inc.

Global Virtual Time (GVT) computation is a key determinant of the efficiency and runtime dynamics of Parallel Discrete Event Simulations (PDES), especially on large-scale parallel platforms. Here, three execution modes of a generalized GVT computation algorithm are studied on high-performance parallel computing systems: (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 to exploit one-sided communication in extant supercomputing platforms. Performance results are presented of implementations of these algorithms on up to 216,000 cores of a Cray XT5 system, exercised on a range of parameters: optimistic and conservative synchronization, fine- to medium-grained event computation, synthetic and nonsynthetic applications, and different lookahead values. Detailed PDES-specific runtime metrics are presented to further the understanding of tightly coupled discrete event dynamics on massively parallel platforms.

Categories and Subject Descriptors: C.2.2 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Protocols*; C.5.1 [**Computer Systems Organization**]: Computer System Implementation—*Large and Medium Computers*; I.6.1 [**Computing Methodologies**]: Simulation and Modeling—*Discrete*; I.6.8 [**Computing Methodologies**]: Simulation and Modeling—*Types of Simulation (Discrete Event, Parallel)*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Parallel discrete event simulation, time warp, global virtual time, one-sided communication, asynchrony

ACM Reference Format:
Kalyan S. Perumalla, Alfred J. Park, and Vinod Tipparaju. 2014. Discrete event execution with one-sided and two-sided GVT algorithms on 216,000 processor cores. ACM Trans. Model. Comput. Simul. 24, 3, Article 16 (June 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2611561

Author's addresses: K. S. Perumalla, 1 Bethel Valley Rd, Oak Ridge, TN 37831-6085; email: perumallaks@ornl.gov; A. J. Park, 1 Microsoft Way, Redmond, WA 98052; email: alfpark@outlook.com; V. Tipparaju, 14231 FM 1464 Rd, Apt 8101, Sugar Land, TX, 77481; email: tipparajuv@gmail.com.
This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy (DOE). 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, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. This research was supported by the Early Career Research Program of the DOE Office of Science, Advanced Scientific Computing Research. The research used resources of the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory, which is supported by the DOE Office of Science.

DOI: http://dx.doi.org/10.1145/2611561

1. INTRODUCTION

Parallel Discrete Event Simulation (PDES) 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. Discrete event execution evolves the states of the underlying entities 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 general, 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. A major challenge is in the design and implementation of efficient algorithms for virtual time synchronization and the verification of the synchronization efficiency at large parallel computing scales on a variety of PDES models.

However, few synchronization algorithms have thus far been gainfully employed on supercomputers with many thousands of processor cores. There is an insufficient understanding about the dynamics of discrete execution on a range of representative applications and benchmarks. Also, advanced network mechanisms, such as one-sided communication of massively parallel platforms, need to be exploited for efficient virtual time synchronization and discrete event execution (related work is discussed in greater detail in Section 5).

The focus of this article is in advancing virtual time synchronization to massively parallel platforms, exploiting specific hardware mechanisms that such large installations offer, and studying the dynamics of discrete event execution. A range of important execution alternatives are studied here to understand their feasibility and efficiency at scale. Mechanisms explored here include synchronous as well as asynchronous execution and the notion of sidedness in terms of conventional "two-sided" communication and the "one-sided" communication natively supported on advanced parallel systems.

1.1. Global Virtual Time

In PDES, independent Logical Processes (LPs) hold encapsulated states, 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 (mixed) approaches.

At the core of execution of PDES engines of all types is the parallel/distributed synchronization of virtual time to correctly process the events in conservative or optimistic fashion. Fast virtual time synchronization algorithms rapidly compute a quantity called the *Global Virtual Time* (GVT) to directly speed up the distributed wave of progress of all processors executing events along the global virtual timeline. The fine-grained nature of event execution imposes tight constraints on GVT algorithms with respect to scalability and speed. Thus, a performance-critical aspect of any PDES engine is the specific GVT computation algorithm it employs.

Multiple, largely equivalent definitions of GVT are possible; see Gomes et al. [1998] and Fujimoto [1999] for surveys. Here, we employ one such view in which GVT is 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 event processing until T_{min} without fear of data dependency

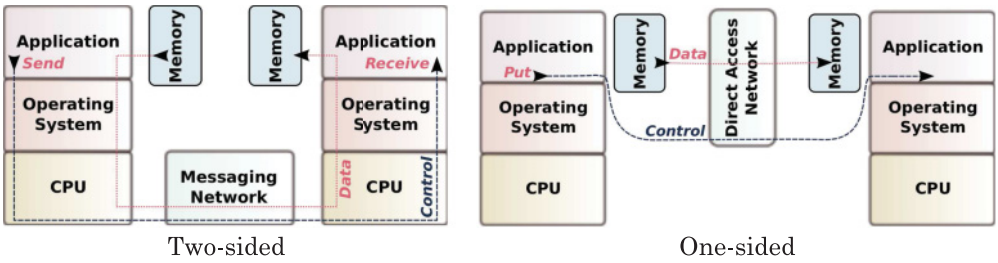

Fig. 1. Functional view of two-sided vs. one-sided communication systems.

violations. Clearly, the rapidity with which T_{min} can be advanced globally has a direct bearing on the speed with which processors can concurrently execute their events.

1.2. Two-sided *vs.* One-sided Communication

In relation to GVT computation on massively parallel systems, GVT algorithms must take into account a communication concept called *sidedness* that exhibits important systems-level effects at scale. Conventional message passing communication on distributed memory platforms falls in the category of "two-sided" communication, while a more direct, memory-to-memory interface between processors is supported in "one-sided" communication. A functional view of the two paradigms is shown in Figure 1.

In two-sided communication, both the receiver and sender sides of the application participate in every data exchange. Two-sided communication essentially requires both sides of the exchange to coordinate for any data transmission. On the other hand, one-sided communication provides a more direct-transfer interface in which a copy of the data from a memory location of the sender is sent to another memory location of the receiver on another processor. Importantly, such a transfer can be performed by the sender *without the participation of the receiver* to complete (and make record of) the transfer. Control information regarding the interprocessor mappings among memory locations and the signaling of events such as start and end of transfer is also transmitted asynchronously via the direct access network.

Our focus here is on using actual, natively supported implementations of one-sided communication interfaces on massively parallel platforms, such as the `Portals` implementation on a Cray XT5 machine (explained later in detail). By contrast, when one-sided communication *interfaces* are supported over two-sided *implementations*, they are not truly one-sided in actual execution. For example, the Message Passing Interface (MPI) standard provides interface routines such as `MPI_Get()` and `MPI_Put()` that are one-sided in semantics, but MPI does not guarantee actual one-sided implementation of those routines. Hence, the one-sided GVT algorithm presented here works best over natively supported one-sided communication implementations.

In almost all parallel systems that support one-sided communication, two-sided communication is also provided as an additional interface that is implemented over either a dedicated fraction of the one-sided communication network or an entirely separate network. Due to this facility, with one-sided communication, GVT information can be exchanged over the one-sided network separately from event data exchanged over the two-sided network, thereby separating the two distinct use cases. The potential advantages of one-sided messaging are: (1) GVT messaging is separated from event communication, thereby eliminating competition and its resultant latency increase for GVT messages and (2) overheads of dynamic memory remapping are avoided due to static inter-processor messaging structure for GVT messages.

In general, it is possible to route event messages also via one-sided communication. However, mixing GVT messages and event messages on the same one-sided network channels can reduce performance because of the fundamental differences between GVT and event messages: (1) GVT messages are short-sized and fixed-sized, which make them well-suited for one-sided communication, whereas event messages are typically larger sized and variable in length and message counts, which significantly complicates one-sided memory buffer allocation; (2) the interprocessor communication structure of GVT messages is fixed and sparse, which makes it easier to configure one-sided control messages, whereas that of event messaging is dynamically variable and potentially dense, which can introduce increase in latency due to dynamic interprocessor configuration; and (3) GVT messages are more constraining on the simulation progress, which makes it better to dedicate the low latency one-sided channels to GVT messaging, shielding them from direct competition/congestion with event messages. Thus, here we focus one-sided communication employed solely for GVT messaging.

1.3. Organization

Due to the complex interactions among model behaviors, hardware features, and software characteristics, the actual scalability and efficiency of any GVT algorithm can only be properly evaluated with actual implementation and benchmarking of PDES engines and applications at scale. To evaluate our GVT algorithms, we study their performance along four different dimensions in PDES application characteristics:

(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 some local events can be processed beyond GVT,
(3) lookahead, which is a measure of static concurrency available in the application scenario, and
(4) interprocessor messaging types, categorized here as two-sided and one-sided.

The rest of the article is organized as follows. The GVT algorithms are described in Section 2, and their implementation details are presented in Section 3. A detailed performance study on a variety of PDES benchmarks is described in Section 4. Prior, related work on virtual time synchronization algorithms is covered in Section 5. The article is concluded and potential future work is identified in Section 6.

2. GVT ALGORITHMS

In a typical PDES execution, the execution engine operates in a loop to process local events (main loop) and also participates in interprocessor synchronization for GVT. The GVT computation, in general, is performed in a separate module (GVT loop) that may be inlined within the main loop or executed in its own thread. Based 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 computation 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. Fast GVT advancement can improve caching behavior because it can reduce the size of the working set by quickly committing, reclaiming, and reusing a small number of memory buffers for events.

2.1. Execution Modes

Here, we focus on three execution modes of a generalized GVT algorithm, covering the space of synchronous versus asynchronous execution and two-sided versus one-sided

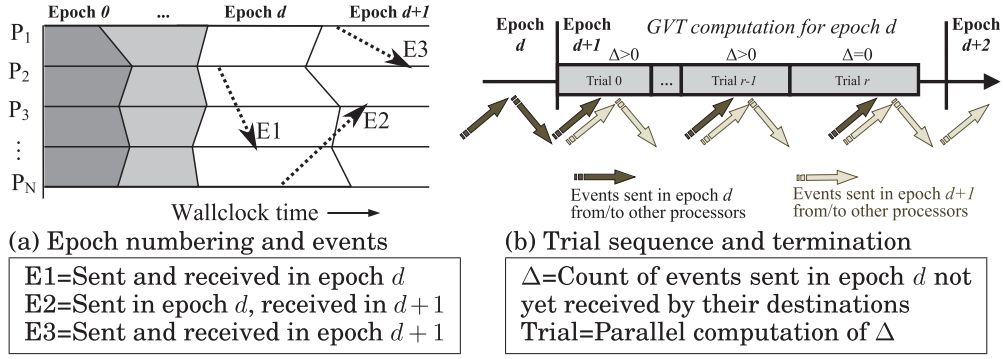

Fig. 2. Distributed snapshots-based GVT computation.

communication. With two-sided communication, both synchronous and asynchronous execution are possible. With one-sided communication, only asynchronous execution is sensible due to its primary benefit, that being the overlap of a receiver's computation with communication. All modes support both conservative as well as optimistic execution.

1 **Two-sided Synchronous**: Whenever the engine initiates a GVT computation, it blocks until the computation terminates and the new GVT value is obtained from it.
2a **Two-sided Asynchronous**: In this mode, the GVT computation and the engine's main loop are concurrently active. Two-sided interprocessor communication is used for event exchanges, as well as for GVT messages.
2b **One-sided Asynchronous**: Just as in 2a, GVT and event loops are concurrent, but they use separate communication mechanisms. Event data are exchanged via two-sided communication, while the GVT is computed via one-sided communication.

All the modes are expressed in a combined algorithmic template, as described next.

2.2. Distributed Snapshots-based GVT Computation

Our unified GVT algorithmic template is based on the general approach of computing distributed snapshots [Mattern 1993; Choe and Tropper 1998]. In this approach, PDES execution is divided into epochs, as illustrated in Figure 2. Each epoch is a distributed snapshot such that no event "goes backward" across epochs. In other words, for every event sent by a processor in epoch numbered d, the event is only received at the destination processor in the same epoch d or a later epoch $d' > d$, but never in a previous epoch $d' < d$. For example, in Figure 2 illustrating valid snapshots, event **E1** is entirely contained in epoch d, whereas **E2** crosses epoch d into $d+1$, and **E3** is contained within epoch $d+1$.

In our algorithm, it is guaranteed that all events sent in epoch d are fully received at their destinations in epochs d or $d+1$, but never beyond $d+1$. This is achieved by ensuring that the global number, Δ, of transient events sent in d yet to be received by their destinations is equal to zero. Events still in flight (sent but not received) are indicated by $\Delta > 0$. The GVT computation is designed to carefully demarcate epoch boundaries at each processor such that they constitute a distributed snapshot while also computing the least event timestamp across all processors.

2.3. Unified GVT Algorithm Template for Conservative and Optimistic Execution

Algorithm 1 shows the pseudocode of our unified GVT algorithmic template. It is parametrized by boolean flags *synchronous* and *conservative* to denote synchronous *vs*.

asynchronous execution and conservative *vs*. optimistic execution, respectively. The template includes four procedures: (1) the main simulation loop $\mathcal{ML}$, (2) the GVT computation loop $\mathcal{GL}$, (3) the procedure $\mathcal{IE}$ that performs accounting for every incoming interprocessor event, and (4) the procedure $\mathcal{OE}$ that tags every outgoing interprocessor event.

ALGORITHM 1: Unified GVT algorithmic template for conservative or optimistic synchronization, and 1-sided or 2-sided messaging, executed at every processor p

Variables

Name		Initial Value	Description
synchronous	$\leftarrow$	user-defined	Is synchronous execution desired?
conservative	$\leftarrow$	system-defined	Is the main loop conservative?
active	$\leftarrow$	**false**	Is a new GVT value being computed?
d	$\leftarrow$	0	Current GVT epoch number
$\delta_p[d]$	$\leftarrow$	0 for all $d \geq 0$	p's contribution to the total count of transient messages sent in epoch d across all processors
$\tau_p[d]$	$\leftarrow$	∞ for all $d \geq 0$	Lowerbound on any event timestamp in epoch d sendable from p and receivable by any processor
$E(T_E)$	$\leftarrow$		An event E with receive timestamp T_E
LVT_p	$\leftarrow$	0	Least of all event timestamps $\{T_E\}$ at p (all local *unexecuted* events if conservative; all local *unprocessed* events and *anti-messages* if optimistic)
LA	$\leftarrow$	user-defined ≥ 0	Inter-processor event lookahead

$\mathcal{ML}$: Main Loop

```
1: while GVT < end time do
2:    while GVT < LVT_p and (not active) do
3:       active ← true
4:       if synchronous or conservative then
5:          Wait until not active
6:       end if
7:       Update LVT_p
8:    end while
9:    if conservative then
10:      Execute some/all E(T_E), T_E ≤ GVT
11:   else
12:      Commit some/all E(T_E), T_E ≤ GVT
13:      and perform rollbacks, if any
14:      or execute some E(T_E), T_E > GVT
15:   end if
16: end while
```

$\mathcal{GL}$: GVT Loop

```
1: start: Wait until active
2: d' ← d
3: d ← d + 1 {Lay cut point}
4: r ← 0
5: τ_p[d'] ← min(τ_p[d'], LVT_p + LA)
6: repeat
7:    Δ ← Σ_{q=1}^{N} δ_q[d']      } Composite
      T ← min_{q=1}^{N} τ_q[d']    } reduction
8:    r ← r + 1
9: until Δ = 0
10: GVT ← T
11: active ← false
12: goto start
```

$\mathcal{IE}$: Handle Incoming Event $E(T_E, d_E)$

```
1: δ_p[d_E] ← δ_p[d_E] − 1
2: if active then
3:    τ_p[d_E] ← min(τ_p[d_E], T_E + LA)
4: else
5:    LVT_p ← min(LVT_p, T_E)
6: end if
```

$\mathcal{OE}$: Tag Outgoing Event $E(T_E)$

```
1: δ_p[d] ← δ_p[d] + 1
2: Tag E as E(T_E, d)
```

A variable d is used as a counter of the number of GVT computations performed so far, which is the same as the current epoch number. Each GVT computation proceeds

as sequence of *trials*, which are successive reductions to determine the number Δ of transient events "in flight." The trials are counted by the variable r ($\mathcal{G}L$ line 8), starting at 0 for each epoch d. The transient event count is computed as a global reduction using an addition operator on the difference between the number of events sent in the previous epoch and the number received in the previous or current epoch. Together with this summation, a global minimum reduction operator is also applied on the local minimum timestamps at each processor. This combined reduction of transient message count and the minimum timestamp is indicated as *composite reduction* in line 7 of $\mathcal{G}L$. When Δ becomes zero, the globally reduced minimum time is usable as a (nondecreasing) GVT value (line 10 of GVT loop $\mathcal{G}L$). If Δ is nonzero, another reduction is started to determine if there has been progress in event delivery that would reduce Δ.

In optimistic discrete event executions, retractions (antimessages) are treated as regular events by using their timestamps, similar to those for regular messages.

The data structures $\delta_p[d]$ and $\tau_p[d]$ are shown to be arrays of length equal to the number of GVT computations. The lengths of the arrays are shown this way for simplicity and clarity of understanding. In practice, the arrays need only have two elements each, and all references by index d can be safely replaced by $d \mod 2$. Thus, every reference to $\delta_p[d]$ is replaced by $\delta_p[d \mod 2]$ and every $\tau_p[d]$ by $\tau_p[d \mod 2]$ in the implementation of $\mathcal{G}L$, $\mathcal{I}E$, and $\mathcal{O}E$ portions of Algorithm 1. This works correctly because no event spans more than two epochs. The algorithm ensures that every event $E(T_E, d_E)$ tagged by the source processor in epoch d_E is received by the destination processor only in epochs d_E or d_{E+1}.

2.4. Correctness of GVT Computation

Here, we present the correctness conditions for the GVT computation and outline a proof sketch for the correctness of the algorithm. Consider a globally frozen snapshot of the PDES execution. The GVT value can be easily computed as the minimum among the timestamps of the events at all processors and the events in flight within the network. However, this "ideal" value GVT cannot be efficiently computed in practice because execution cannot be frozen precisely at the same time on a large number of processors. Instead, the GVT algorithms compute an estimate $\widetilde{GVT} \leq GVT$ that is always bounded by the ideal GVT value (in the case of multiple, concurrently active GVT computations, the largest computed value is used as the $\widetilde{GVT}$). The challenge is to advance $\widetilde{GVT}$ as fast and as close to GVT as possible without violating the properties of GVT. In particular, the properties of the ideal GVT value must be retained: $\widetilde{GVT}$ should never regress, and no processor should ever receive an event with a timestamp less than $\widetilde{GVT}$. These requirements translate to the following important correctness conditions in the implementation of GVT computation (see, e.g., Fujimoto and Hybinette [1997] and Holder and Carothers [2008]).

—**Transient messages**: For any epoch, a transient message is an event in flight (sent but not yet received) that needs to be accounted for in its epoch. In other words, the timestamp T_E of every such event $E(d_E, T_E)$ in the network must be included in the GVT computed in epoch d_E. Otherwise, the GVT value can regress because failure to account for E can take $\widetilde{GVT}$ further than safety ($T_E < \widetilde{GVT}$) when E eventually arrives at its destination. This violation of correctness is avoided by rejecting all candidates for $\widetilde{GVT}$ when even a single transient event exists in the network. Finally, when no transient event exists, all events are present in processor memories, and hence their timestamps are all included in the global minimum. The algorithm uses

this approach in the GVT loop (line 6 to line 9) by rejecting all candidate $\widetilde{GVT}$ until no transient events exist as indicated by $\Delta = 0$.

—**Simultaneous Reporting**: The GVT computation must also ensure that every event always has at least one processor that takes ownership of the timestamp of the event insofar as accounting for its timestamp. This is achieved by a combination of two conditions: the atomicity of the establishment of the cut point line 3 and the fact that no transient messages exist when $\widetilde{GVT}$ is evaluated at line 7.

A more rigorous proof of correctness can be derived using proof by induction for the transient message problem and proof by contradiction for the simultaneous reporting problem, along the lines of the proofs in the GVT literature (e.g., Holder and Carothers [2008]).

2.5. Synchronous Two-sided GVT

The synchronous two-sided execution mode is achieved in Algorithm 1 by setting the *synchronous* variable to **true**, and using a *blocking* reduction operation (such as the `MPI_Allreduce()` routine of MPI) performed in the GVT loop $\mathcal{GL}$ at line 7. This mode is a generalization of previous algorithms in the literature [Perumalla and Fujimoto 2001; Holder and Carothers 2008; Bauer Jr. et al. 2009], enhanced to support conservative as well as optimistic execution *with lookahead*.

Although this execution mode is relatively easy to implement (e.g., using the blocking collectives of MPI), the blocking nature requires every processor to stop processing its events while the GVT is being computed. It also is prevented from sending any additional events (as part of executing local events) to other processors. In conservative execution, in order to prevent other processors from blocking for too long, every processor must join the GVT computation periodically, even if that processor itself locally has events to safely process. In optimistic execution, processors must quit optimistic event processing while being blocked. Thus, blocking in both modes is detrimental, especially since the blocked time increases with the number of processors. On massively parallel platforms, the blocked time can grow substantially. Note that hardware-accelerated collectives (e.g., Blue Gene collective networks [Almási et al. 2005; Faraj et al. 2009]) help only a little in decreasing the time taken for the collectives because the blocked time is dominated by the time difference between the first and last joining processors, which cannot be accelerated by hardware. Nevertheless, this synchronous algorithm can work well for well-balanced work loads in which the event timestamp distribution is relatively uniformly spread across all processor timelines.

2.6. Asynchronous Two-sided GVT

The asynchronous two-sided execution mode is achieved in Algorithm 1 by setting the *synchronous* variable to **false** and using a *nonblocking* global reduction operation performed in the GVT loop $\mathcal{GL}$ at line 7.

We implemented an optimized nonblocking reduction operator over the two-sided MPI point-to-point messaging primitives such as `MPI_Isend()`, `MPI_Irecv()`. This asynchronous, application-level reduction is performed using a tree topology that uses a butterfly communication pattern among all nodes at the node level, in which only one core (core 0) per node participates. Each core 0 communicates internally with other cores on its node using a centralized communication topology, thus minimizing network traffic and making effective use of shared memory communication within the node.

In this execution mode, note that the processor starts and leaves an active GVT computation in the main loop $\mathcal{ML}$ at line 3, thereby avoiding blocking. While GVT is being computed, it continues to execute any additional local events that are processable (safe

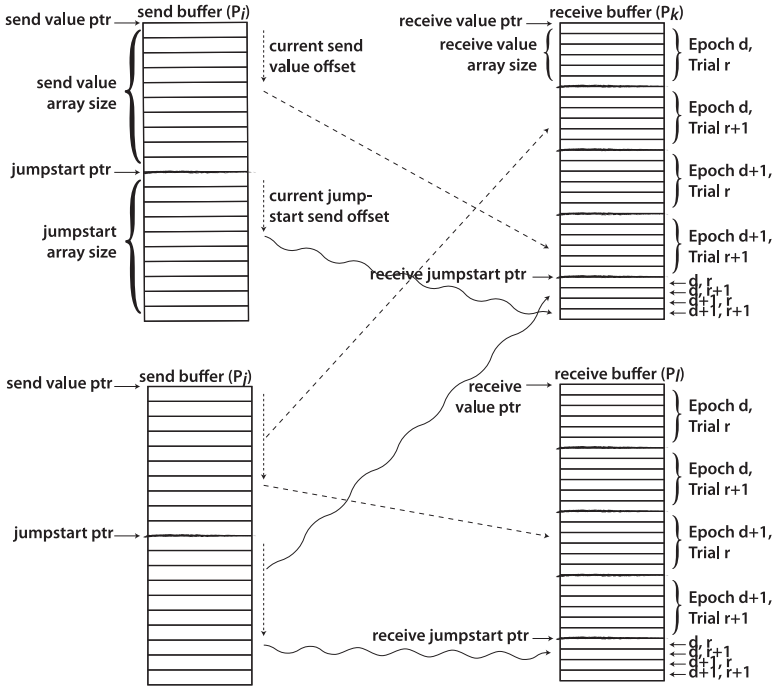

Fig. 3. Data structures for one-sided asynchronous GVT computation. Each in the send or receive buffers includes the $(LVT_p + LA, \delta_p)$ values of the corresponding sender p. Dashed lines represent put operations of reduced values. Squiggly lines represent put operations of *jump start* messages to initiate reduction at the receiver.

events in conservative execution or future events in optimistic execution). Additionally, it is also free to send and receive events without any restrictions, unlike in the previous synchronous two-sided algorithm.

2.7. Asynchronous One-sided GVT

The one-sided GVT operates similarly to the asynchronous two-sided GVT, with one major difference: the reduction operations are performed asynchronously using direct-memory operations on remote processors, with data transfer carried out asynchronously by the network. This achieves nonblocking operation for GVT messages in a way that is completely decoupled from event messaging. Normally (as in the previous two-sided asynchronous algorithm), nonblocking GVT must perform its synchronization via messaging that is 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. On the other hand, assuming *efficient* implementations of one-sided communication on the parallel machine, GVT messages can be exchanged with minimal delay by decoupling them from the event communication.

The one-sided GVT algorithm must arrange memory buffers across processors in such a way that asynchronous reductions at line 7 are all performed using direct memory-to-memory assignment across processors. To enable such operation, the memory organization at each processor is shown in Figure 3, 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.

The asynchronous reductions are performed using the same interprocessor exchange structure as for the asynchronous two-sided GVT mode. With the same tree topology optimized for hierarchical reductions on multicore architectures, the interprocessor 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, which contains the tuple $< P_{source}, d, r, LVT_{source}, \delta >$, where P_{source} is the sending processor and LVT_{source} is the minimum local timestamp at P_{source}. Additionally, room for *jumpstart* messages is also allocated such that processors may jumpstart other processors (within or outside their 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.

3. IMPLEMENTATION

We now present implementation details of the algorithms incorporated into the μsik discrete event execution engine [Perumalla 2005; Perumalla et al. 2011]. The μsik PDES engine is based on a "micro-kernel" approach to support a variety of execution modes including conservative, optimistic, and hybrid execution. It uses Time Warp-style execution for the optimistic mode that is both risky and aggressive (local optimistic execution and transmission of optimistic events to other processors). Support for both primary and secondary rollbacks are implemented. Rollback of logical process state is based on reverse computation (although state saving is also implemented, it is not exercised in the applications used for the study).

In μsik, a new GVT computation is always initiated as soon as a previous GVT completes to minimize blocking for conservative LPs and to minimize uncommitted activity for optimistic LPs. All the GVT algorithm execution modes have been implemented into μsik, any one of which can be chosen by the user at runtime initialization via an environment variable specification. Both nonblocking and one-sided GVT algorithms are carefully implemented such that no barriers are invoked from the main loop. All the benchmarks used here to evaluate the GVT algorithm performance are written as applications over μsik.

The implementation and experimentation were performed on a Cray XT5 system with 18,688 nodes, in which each node consists of two 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 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. All interprocessor event communication is performed using traditional two-sided communication via the MPI. The GVT message exchange for two-sided GVT algorithms is also performed using MPI. Asynchrony with respect to event messaging is realized via `MPI_Iprobe()` for two-sided communication. 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

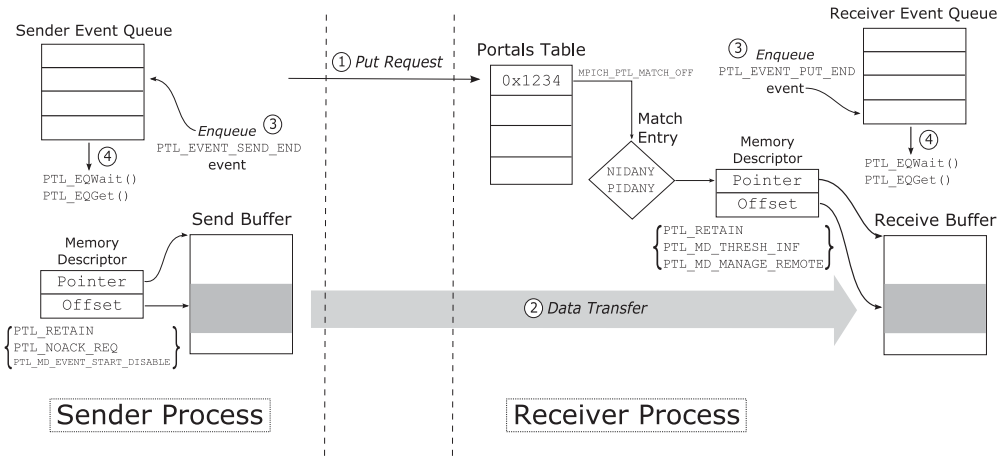

Fig. 4. Portals data structures and data movement operations for one-sided GVT.

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 the previously described (optimized butterfly) topology, using MPI messaging for exchanging the reduction control messages.

For one-sided GVT algorithms, the Portals interface [Brightwell et al. 2005] is used for one-sided communication. The Portals API on the Cray XT5 is implemented using the 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. A Direct Memory Access (DMA) program to send/receive data from/to each Portals Memory Descriptor (MD) is generated in the host and transferred to the SeaStar DMA queue. On SeaStar, message transmission and reception machinery each utilize a single first-in, first-out (FIFO) and a single Direct Memory Access (DMA) queue. Small messages (<16 bytes) are handled directly from the FIFO, while larger messages utilize the DMA Queue. The message transmission machinery on SeaStar has a 32-entry Transmit (TX) DMA queue. Elements in the receive engine machinery in SeaStar NIC are (a) 256-entry Receiver (RX) DMA queue that includes a DMA program in each entry for DMA into a specific memory descriptor, (b) a 256-entry Content Addressable Memory (CAM) table that maps incoming messages to one of the RX DMA queue entry, and (c) an interrupt mechanism that interrupts the host when such a match cannot be found in the CAM table. If a particular host receives messages from more than 256 different sources and continues to do so randomly throughout the life of the program, the number of host interruptions increases significantly. This is detrimental to both the application running at the receiver (because its computations are interrupted) and the sender (due to higher latency incurred by the interrupt).

The functionality of the one-sided GVT algorithm implementation is illustrated in Figure 4. Portals is initialized with 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 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.

The order in which one-sided communication proceeds in the GVT algorithm is indicated by the circled numbers in Figure 4. The send buffer in this figure corresponds to a send buffer shown on the left side of Figure 3; similarly, the receive buffer in this figure corresponds to a receive buffer shown on the right side of Figure 3. Operations tagged with the same circled number in Figure 4 indicate concurrent operation across the sending and receiving processors.

4. PERFORMANCE ANALYSIS

We examine the dynamics of discrete event execution exercised with the GVT algorithms presented in this work. The performance study is intended to benefit users and researchers of discrete event execution in terms of the scales and speeds that can be expected for various complex combinations of model-level and system-level parameters. In order to evaluate the efficacy of each GVT algorithm, we selected four PDES benchmarks that represent a wide cross-section of PDES application characteristics, from varied event densities to mixed messaging and event computation intensities.

4.1. Execution Benchmarks

We use the following four PDES applications that run over *μsik*. The applications automatically inherit the runtime choice of functionality of all the three GVT algorithm implementations and their optimizations incorporated into *μsik*.

4.1.1. RCPHOLD. The PHOLD application is a *de facto* standard PDES benchmark commonly 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 in 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 (σ, γ), where σ is the number of LPs per core, and γ is a specific parameter for the simulation. For PHOLD, γ 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 ω cores.

The "RC" moniker of RCPHOLD stands for reverse computation. Rollback in optimistic simulation of PHOLD is obtained via reverse computation instead of state-saving. In other words, instead of saving and restoring the simulation state prior to every event, the simulator performs a sequence of undo computations only for rolled-back events to restore the simulation state.

4.1.2. RCREDIF. Another PDES benchmark used is called RCREDIF, which is a large-scale epidemiological outbreak simulation based on a reaction-diffusion model [Perumalla and Seal 2011]. 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 (γ), a number of locations per region (σ), 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 in both conservative mode as well as optimistic mode using reverse computation, and it 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.

4.1.3. $\mu\pi$. $\mu\pi$ is a software-based experimentation platform for testing synthetic and real unmodified MPI programs [Perumalla 2010; Perumalla and Park 2011]. $\mu\pi$ multiplexes virtual MPI ranks per real rank (the ratio to be referred to as *LPX*) for execution over simulated virtual platforms through μsik's process-oriented PDES framework. An MPI rank is the unique integer assigned to each task (typically, a UNIX process) in a parallel application based on MPI. Each virtual MPI rank is the simulated counterpart of the real MPI rank that appears in the simulated scenario.

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.

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.

4.2. Experiment Setup

The GVT algorithms were tested with all the aforementioned applications. For labels in all of the following charts, we use a 3-tuple "*XYZ*" to identify the scenario tested, as follows:

	Description	Values
X	Synchronization strategy	`C` for conservative or `O` for optimistic
Y	GVT communication "sidedness"	`1` for one-sided, `2` for two-sided
Z	Execution mode	`S` for synchronous, `A` for asynchronous

Table I. Notation Used in Charts

	Description
ε	Aggregate committed event rate in millions of events simulated per wall clock second
λ	Number of GVT epochs
ρ	Maximum number of rollbacks observed on a single core
α	Average rollbacks per core

	Description
(σ, γ)	Structure of simulation
C..	Conservative simulation
O..	Optimistic simulation
.2S	Two-sided synchronous GVT
.2A	Two-sided asynchronous GVT
.1A	One-sided asynchronous GVT

For example, O2A refers to the optimistic execution of two-sided asynchronous GVT algorithm, and C1A refers to the conservative one-sided asynchronous GVT algorithm.

Combinations of lookahead, structure, synchronization strategy, and GVT algorithms were varied for each benchmark. For the RCPHOLD and RCREDIF benchmarks, a range of lookahead values were exercised: 0.01 (very low), 0.1 (low), 0.5 (medium), and 1.0 (high).

Additionally, the structure (σ, γ) of each application was varied between (10,1000) and (100,100). Thus the message population remained constant between structures per core. The number of LPs per core is varied, resulting in high and low event densities per LP for the respective scenarios. The number of remote messages remains the same per core but is an order of magnitude more per LP in the (10,1000) case. The results from the structure of (10,1000) are shown here. Additional performance data for more lookahead values and for the structure of (100,100) are included in Section A of the online supplement.

For $\mu\pi$ benchmarks, lookahead was fixed based on the network properties. Here, we selected prototypical values of "fast" (i.e., latency of 10μs and bandwidth of 1Gb/s) and "very fast" (i.e., latency of 1μs and bandwidth of 10Gb/s) network specifications to determine the lookahead. The "structure" of the $\mu\pi$ benchmarks is specified by *LPX* (i.e., number of virtual MPI ranks multiplexed on each real rank), where values of 128 and 1,024 were chosen to exercise light and very heavy multiplexing.

The simulation end times were set to 1,000 simulated seconds for all RCPHOLD runs and 168 simulated hours (1 week of disease spread = 24 hours/day $\times$ 7 days) for all RCREDIF runs. $\mu\pi$ barrier test simulated one virtual barrier for all *LPX*, whereas $\mu\pi$ ping test simulated up to 1KiB and 64KiB of data transfer in the $LPX = 1{,}024$ and $LPX = 128$ structures, respectively.

In the following charts, performance is plotted as event rate or actual runtime. The ordinate labeled ε denotes the aggregate committed event rate in millions of events per second. Each individual data point for the three GVT algorithms is plotted while the best performing GVT algorithm (i.e., the algorithm achieving the highest ε) is noted at each core count. A line joining the maxima is drawn through these best points to visualize the scaling trendline of performance.

The charts for RCPHOLD and RCREDIF include λ, which denotes the number of GVT epochs. In the charts for optimistic runs, ρ denotes the maximum number of rollbacks occurring on a single core within the entire simulation, and α is the average number of rollbacks per LP.

The symbols and notations are summarized in Table I.

4.3. Performance Results

The performance results for the GVT algorithms are shown in the following charts:

—RCPHOLD results are shown in Figure 5 and Figure 6 for conservative execution, and in Figure 7 and Figure 8 for optimistic execution. Additional performance data are shown in the online appendix in Section A.1.

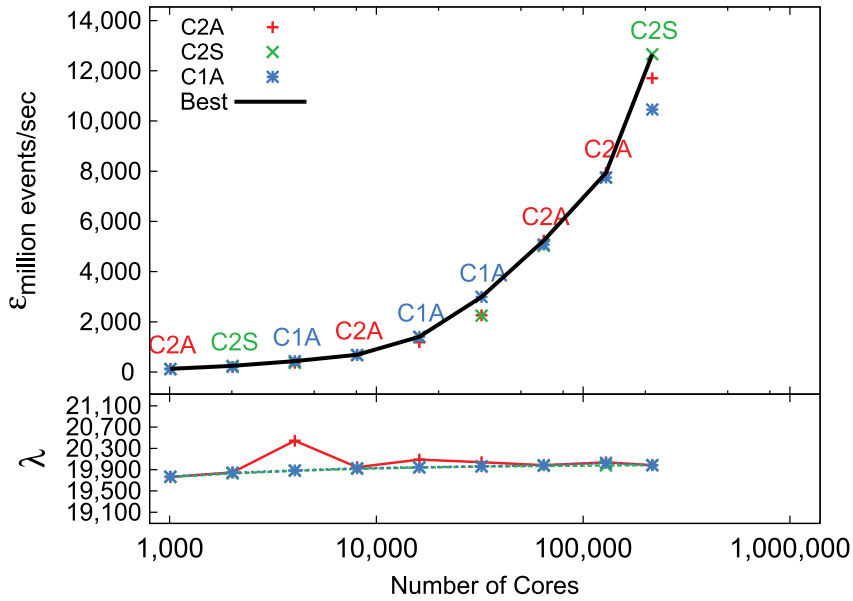

Fig. 5. RCPHOLD conservative, low lookahead LA = 0.1, structure (10,1000).

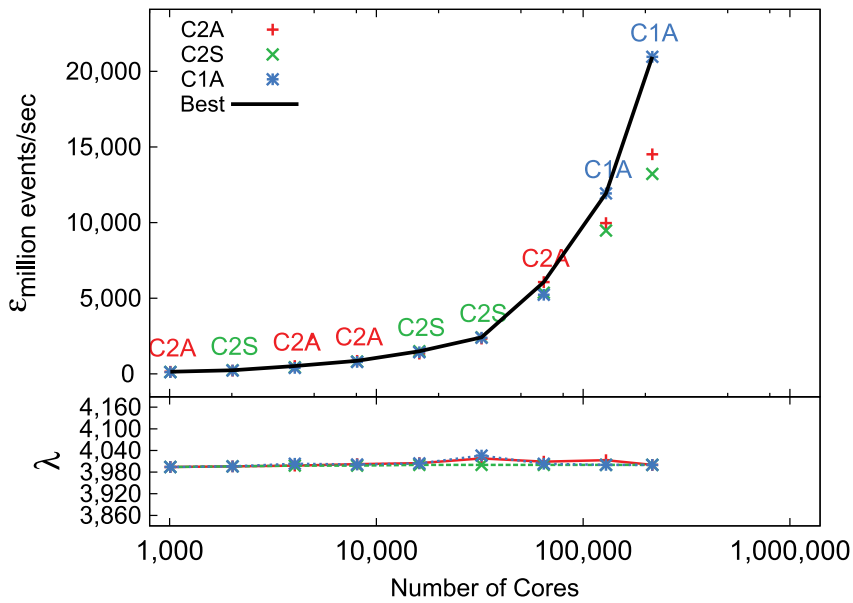

Fig. 6. RCPHOLD conservative, medium lookahead LA = 0.5, structure (10,1000).

—RCREDIF results are shown in Figure 9 and Figure 10 for conservative execution, and in Figure 11 and Figure 12 for optimistic execution. Additional performance data are shown in the online appendix in Section A.3.
—Figure 13 shows the best and worst case speedup for RCPHOLD, and Figure 14 shows the same for RCREDIF.

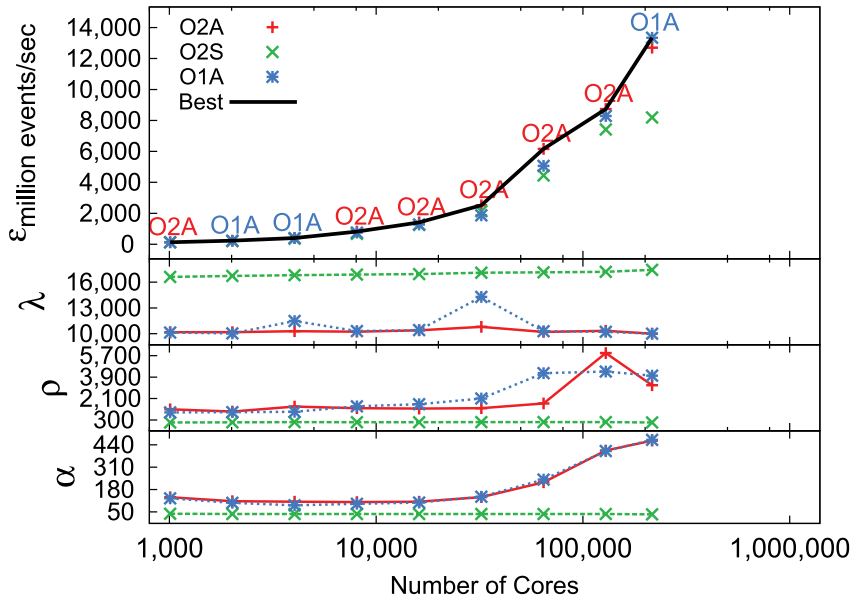

Fig. 7. RCPHOLD optimistic, low lookahead LA = 0.1, structure (10,1000).

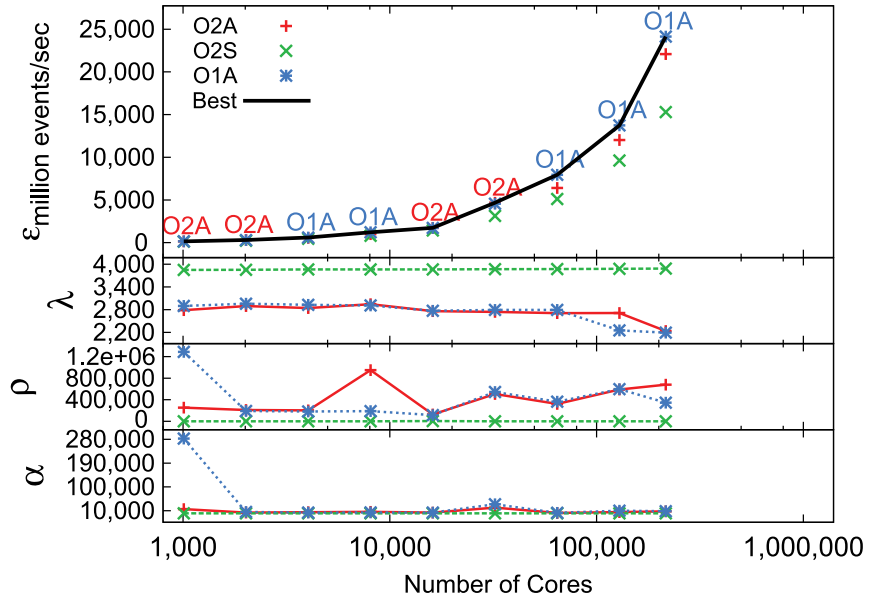

Fig. 8. RCPHOLD optimistic, medium lookahead LA = 0.5, structure (10,1000).

—Figure 15 shows the scaling effects of the GVT algorithms on the $\mu\pi$ Barrier Test benchmark, and Figure 16 show the same for the $\mu\pi$ Ping benchmark. In these process-oriented parallel simulations, the gain from using asychronous GVT algorithms is clearly evident. Process-oriented discrete event execution (exercised with $\mu\pi$), which time-multiplexes multiple thread contexts on a single core, can be seen to benefit the most from asynchronous GVT algorithms at large multiplexing levels

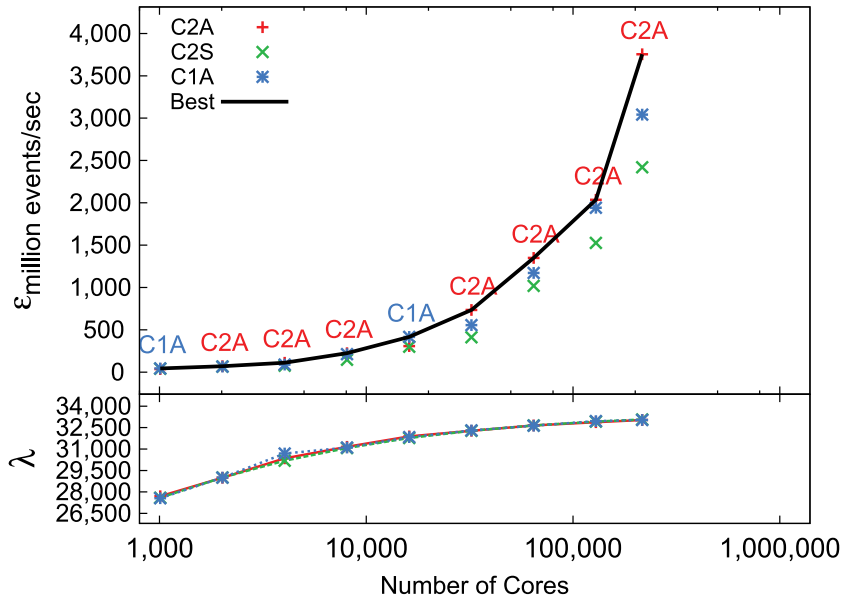

Fig. 9. RCREDIF conservative, very low lookahead LA = 0.01, structure (10,1000).

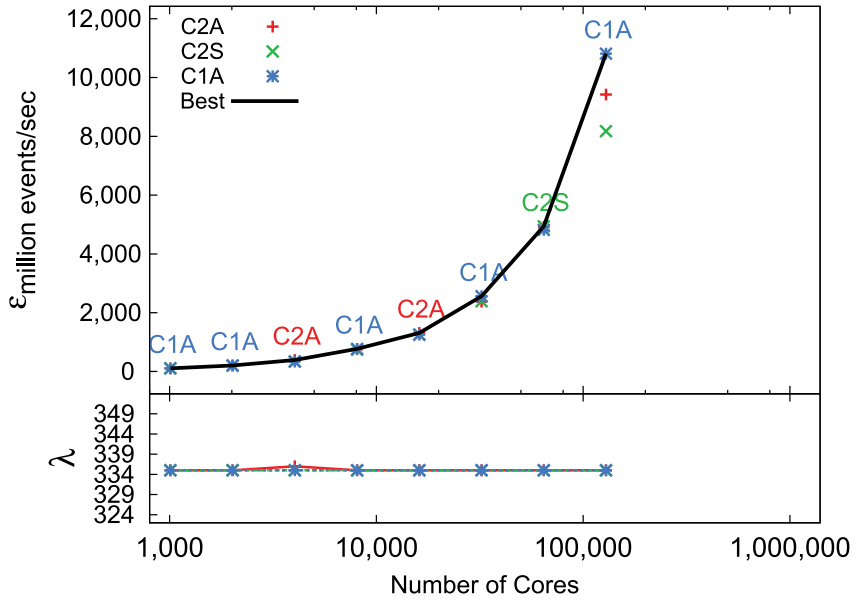

Fig. 10. RCREDIF conservative, high lookahead LA = 1, structure (10,1000).

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 the GVT algorithm appears beneficial in allowing events to be processed concurrently with GVT computation.

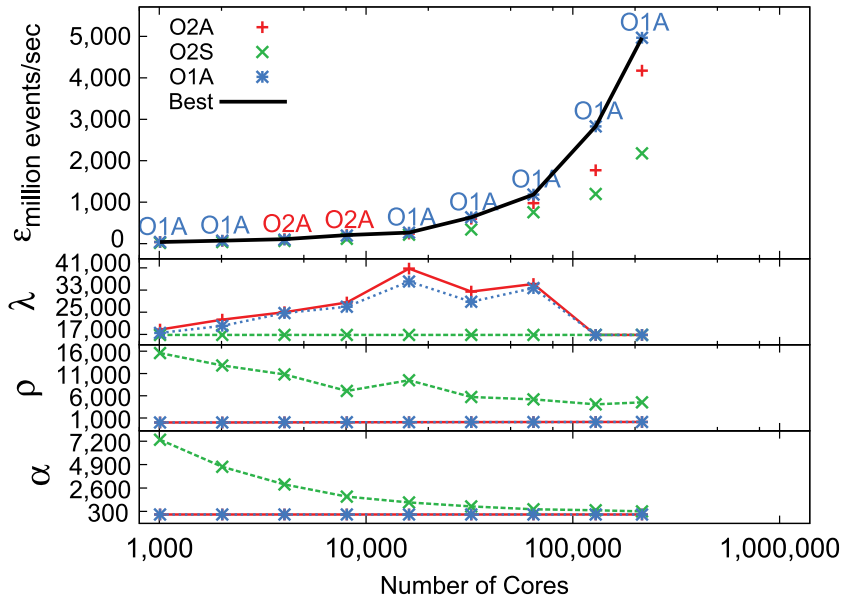

Fig. 11. RCREDIF Optimistic, Very Low Lookahead LA = 0.01, Structure (10,1000).

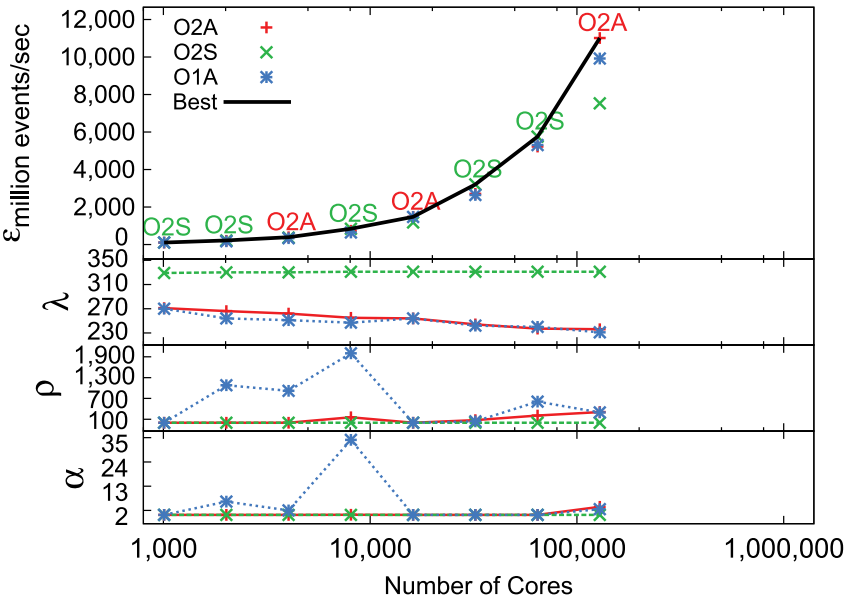

Fig. 12. RCREDIF optimistic, high lookahead LA = 1, structure (10,1000).

4.4. Statistical Variation

All performance experiments were carried out on the (then) top-ranked supercomputer in the world. A significant practical constraint in the use of the largest supercomputers is the limitation on the total amount of allocated time per project for experiments on the machine, especially for benchmarking studies such as the performance study described in this article. Although a large portion of the parameter space has been

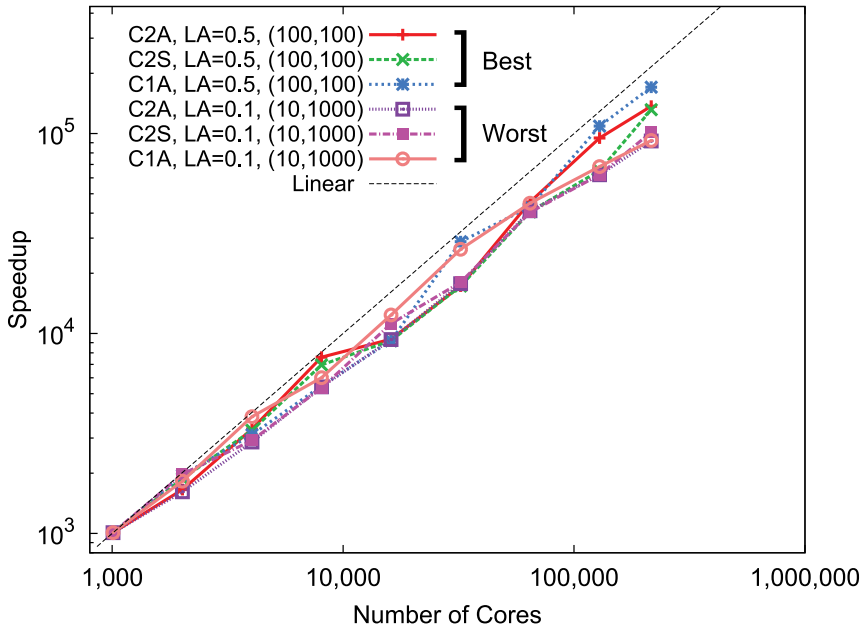

Fig. 13. RCPHOLD speedup for best and worst committed event rates.

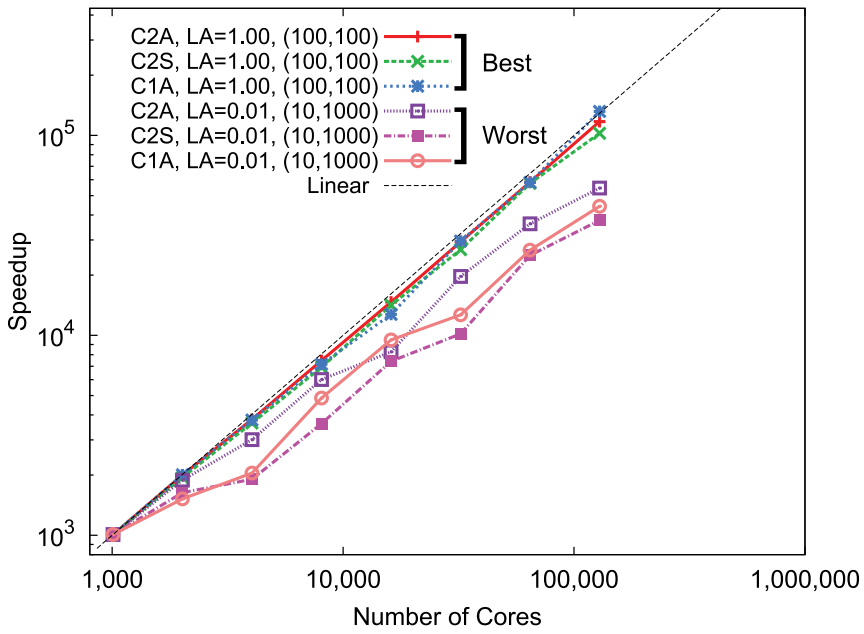

Fig. 14. RCREDIF speedup for best and worst committed event rates.

covered (representing one of the most comprehensive studies of large-scale discrete event execution to date), the limitation on total supercomputing time has some bearing on the statistical significance of the performance data, especially on the largest scale.

To understand the variability of performance data, we have verified the repeatability of performance metrics from 1K up to 128K processors. Multiple runs were made on

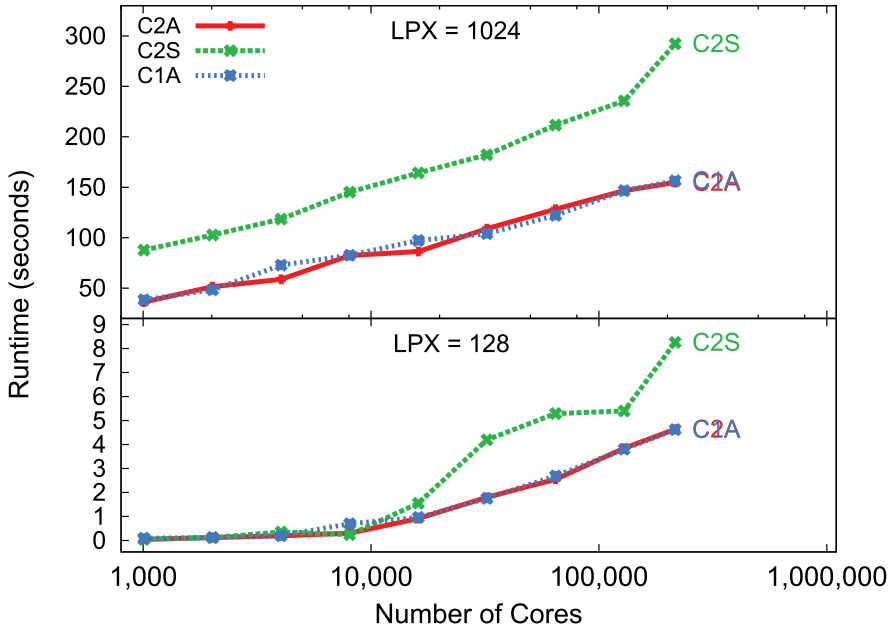

Fig. 15. $\mu\pi$ Barrier test runtime performance.

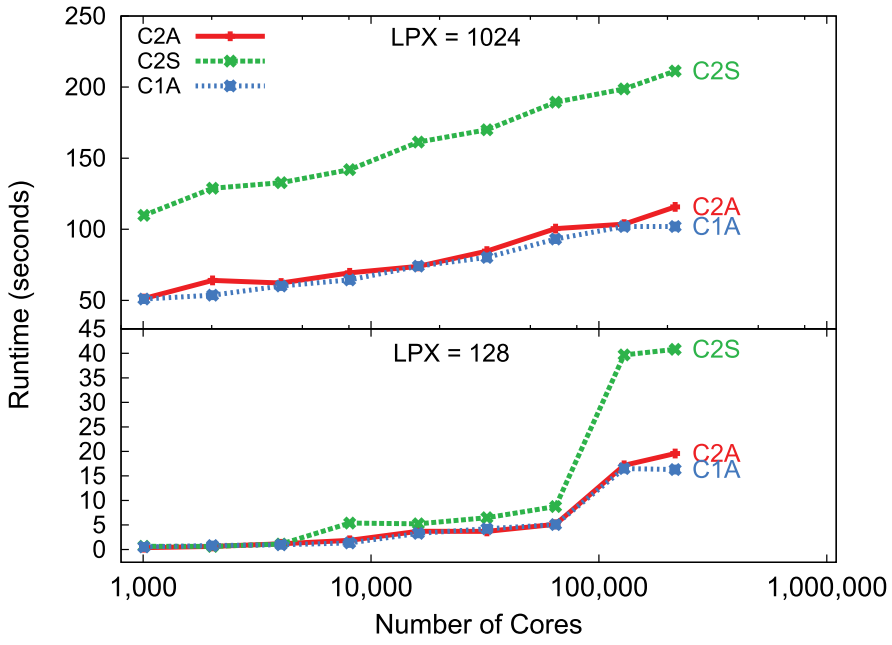

Fig. 16. $\mu\pi$ Ping test runtime performance.

these small- to medium-scale configurations with almost identical timings that are statistically insignificant with respect to variability. This is in line with expectations because the machine, by design, exhibits negligible runtime jitter on small-/medium-scale parallel jobs, but the jitter is not necessarily negligible "at scale" when a single

parallel job uses *all* processor cores of the machine [Skinner and Kramer 2005; Oral et al. 2010]. For the largest configurations (216K processors), the data points in charts represent the performance observed from a single run per data point because each single "at-scale" run consumes an extremely large number of allocated hours, which are very expensive in terms of resources. Hence, comparisons of performance differences between the different GVT variants cannot be reliably done for the large processor counts. Nevertheless, for a high-level understanding, it is observed from the charts that the "best" trendline on the charts exhibits fairly smooth behavior, indicating an overall reliable scaling behavior of the algorithms from 1K up to the largest scale of 216K processors.

5. RELATED WORK

Research in efficient virtual time synchronization experienced rich development over the past few decades. In the early 1990s, after parallel/distributed discrete event simulation began gaining sufficient attention in the research community, a spurt of synchronization algorithms were proposed for use in conservative schemes, optimistic schemes, or both. A rich variety of parameters were considered, such as the presence or absence of FIFO guarantees, message send-back protocols, virtual time horizons and windows, message acknowledgments, and different network architectures or topologies such as token rings and trees. These advancements included seminal works such as the Chandy Misra Bryant (null message) algorithm [Chandy and Misra 1981], the Time Warp algorithm [Jefferson and Sowzral 1982; Jefferson 1985], Samadi's algorithm [Samadi 1985] based on message acknowledgments, Mattern's algorithm based on distributed snapshots [Mattern 1993], Lin-Lazowska algorithm based on combination of timestamped histories and message sequence numbers [Lin and Lazowska 1990], the Bauer-Sporrer algorithm based on centralized resolution on FIFO network channels [Bauer and Sporrer 1992], and many other variants [Reiher et al. 1990; Preiss et al. 1991; Nicol 1993a, 1993b; Lipton and Mizell 1990; Konas and Yew 1992; Felderman and Kleinrock 1991; DeVries 1990]. The majority of them were analyzed for theoretical correctness of operation and analytical complexity measures such as latency between successive time advances [Gomes et al. 1998; Fujimoto 1999].

Additional innovative directions, such as hardware-assisted and shared memory-optimized algorithms, were pursued in the early and mid-1990s. Reynolds et al. [Reynolds et al. 1993] proposed hardware support for GVT computations using a specialized co-processor style of interface to central processing units to offload reduction operations onto the custom-designed co-processors and their interconnection network. Perhaps due to the hardware limitations of their times, performance study was only limited to a few dozen processors. Rosu et al. proposed offloading GVT computations to programmable, general-purpose network cards, again limited in scale by the network hardware technologies of the time [Rosu et al. 1997]. Fujimoto and Hybinette developed a fast algorithm for GVT computation optimized for shared memory machines, carefully avoiding any locking (semaphore) costs among concurrent threads of execution [Fujimoto and Hybinette 1997].

Later, in the late 1990s and early 2000s, synchronization gained additional attention largely in the context of the Time Management interface of the High Level Architecture (HLA) and in the context of large-scale computer network simulations. The former saw real-time (receive ordered) executions scaled to 10^3 processors, but time-constrained executions were limited to 10^2 processors using centralized run-time infrastructures. The latter saw conservative time synchronization advanced to the 10^3-processor scale using global reductions-based algorithms [Fujimoto et al. 2003] as well as null message variants exploiting locality of communication [Park et al. 2004]. Also, virtual time algorithms for operation over combinations of unreliable and reliable network channels

for events and synchronization messages were developed [Perumalla and Fujimoto 2001] and applied to both HLA simulations as well as network simulations.

In mid-2000s, research continued, attempting further scaling [Chen and Szymanski 2005, 2008] using a centralized algorithm on up to 1,000 processor cores. Also, ways to combine real-time with virtual-time advances were explored [McLean et al. 2004; Bauer Jr. et al. 2005]. A GVT algorithm based on "network atomic operations" was designed by Bauer et al., and its performance was evaluated on a small scale of 16 processors [Bauer Jr. et al. 2005].

In the late 2000s, a marked shift occurred, catching up to the dramatically increasing sizes of the supercomputing installations that began to emerge with 10^4 processor cores for the first time. The focus of some of the GVT algorithm design and implementation, as a result, shifted to scalability, aimed at sustaining discrete event execution on some of the largest computing installations. Feasibility of executing conservative, optimistic, and mixed-mode executions on up to 32,768 processor cores was first realized on Blue Gene/L platforms [Perumalla 2007], followed by speed and efficiency improvements to optimistic execution on Blue Gene/L [Holder and Carothers 2008; Bauer Jr. et al. 2009]. Additional analyses of the event dynamics between conservative and optimistic execution were compared with synthetic benchmarks on 16,384 Blue Gene/L cores [Carothers and Perumalla 2010]. In the line of evolution, the next largest supercomputing installations emerged, containing on the order of 10^5–10^6 cores. Execution of PDES at this scale is the focus of this article (a preliminary version of this work appeared in Perumalla et al. [2011]).

6. CONCLUSIONS AND FUTURE WORK

A new unified and parametrized GVT algorithm has been presented that supports synchronous and asynchronous execution, one-sided and two-sided communication mechanisms, and conservative and optimistic synchronization. Within this unified view, three important practical variants were identified. For the variant of asynchronous one-sided operation, a novel implementation based on one-sided communication has been proposed and realized at scale for the first time here.

Using the unified algorithm and its three specific variants, a large parameter space has been covered in a performance study that is the most comprehensive to date in large-scale parallel discrete event simulation. The study included:

—conservative and optimistic modes of synchronization;
—four disparate applications exhibiting dynamics that are very different from each other in terms of the number LPs, the number of events per LP, the event computation granularity, the model lookahead, the inter-LP event exchange structure, the inter-LP event exchange frequency, and the number of event types;
—event-oriented LPs and process-oriented LPs;
—small-, medium-, and large-scale execution, up to 216K processors;
—one-sided and two-sided communication;
—synchronous (blocking) and asynchronous (nonblocking) computation; and
—collection of a wide range of performance-critical data such as event rates, frequencies of GVT computations, and rollback counts.

Detailed quantitative performance data have been documented on up to 216,000 processor cores of a supercomputing system.

—In the fastest RCPHOLD runs, the event rate was nearly 54 billion events per second on 216,000 cores (Figure 18).
—The best self-relative speedup for RCPHOLD exceeded 170,600 on 216,000 cores (Figure 13).

—The best self-relative speedup for RCREDIF exceeded 131,790 on 129,024 cores (Figure 14). The superlinearity is attributable to a suboptimal, cache-limited performance of the baseline run on 1,008 cores.
—In the largest runs of RCREDIF on 216,000 cores, over 2.1 billion individuals were simulated.
—In the barrier test of $\mu\pi$, one-sided GVT gave 1.67× faster execution than with synchronous GVT. The ping test of $\mu\pi$ executed over 2.07× faster with one-sided GVT than with synchronous GVT.
—With $\mu\pi$, a total of 221,184,000 virtual MPI ranks were simulated, representing the largest case of process-oriented parallel discrete event execution to date.

The results have been obtained from a large number of runs with the goal of documenting the effects of performance-critical factors such as lookahead, synchronization mode, and event loads at very large scales. The results are useful to engine developers, application developers, and users by providing them an overview of the dynamics and algorithmic possibilities in large-scale discrete event execution.

The results highlight PDES as a potential candidate in the class of supercomputing applications, one whose benefits can be explored in domains such as Internet simulations, vehicular transportation simulations, and social behavioral simulations. Integration of large-scale agent-based simulation with discrete event execution is part of our future work, along with evaluation with more discrete event applications. Porting and optimizing to the next generation of network interconnects (such as the new networks in multipetascale to exascale systems) remain to be explored. Also of immediate interest is the exploration of the interplay with processor-accelerator hybrid computing technologies that are incorporated into the largest supercomputing platforms. It would also be interesting to explore the use of one-sided communication for event messaging as well, complementing that for GVT algorithm communication.

REFERENCES

G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. 2005. Optimization of MPI collective communication on Bluegene/L systems. In *Annual International Conference on Supercomputing*. ACM, 253–262.

H. Bauer and C. Sporrer. 1992. Distributed logic simulation and an approach to asynchronous GVT-calculation. In *Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 205–208.

D. W. Bauer Jr., C. D. Carothers, and A. Holder. 2009. Scalable time warp on Blue Gene supercomputers. In *Workshop on Principles of Advanced and Distributed Simulation*. 35–44.

D. W., Bauer Jr., G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman. 2005. Seven-o-clock: A new distributed GVT algorithm using network atomic operations. In *Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 39–48.

R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. Underwood. 2005. Implementation and performance of portals 3.3 on the Cray xt3. In *Cluster Computing*. IEEE Computer Society, 1–10.

C. D. Carothers and K. S. Perumalla. 2010. On deciding between conservative and optimistic approaches on massively parallel platforms. In *Winter Simulation Conference*. 678–687.

K. M. Chandy and J. Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Communications fo the ACM* 24, 4, 198–205.

D. Chen and B. K. Szymanski. 2005. DSIM: Scaling time warp to 1,033 processors. In *Winter Simulation Conference*. IEEE, Orlando, FL.

G. G. Chen and B. K. Szymanski. 2008. Time quantum GVT: A scalable computation of the global virtual time in parallel discrete event simulations. *Scalable Computing: Practice and Experience* 8, 4, 423–435.

M. Choe and C. Tropper. 1998. An efficient GVT computation using snapshots. In *Conference on Computer Simulation Methods and Applications, Society for Computer Simulation*. 33–43.

R. C. DeVries. 1990. Reducing null messages in Misra's distributed discrete event simulation method. *IEEE Trans. on Software Engineering* 16, 1, 82–91. January 1990.

A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels. 2009. MPI collective communications on the Blue Gene/P supercomputer: Algorithms and optimizations. In *High Performance Interconnects. IEEE Symposium on*. 63–72.

R. Felderman and L. Kleinrock. 1991. Two processor time warp analysis: Some results on a unifying approach. In *Advances in Parallel and Distributed Simulation*. SCS Simulation Series, vol. 23. 3–10.

R. M. Fujimoto. 1999. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, New York.

R. M. Fujimoto and M. Hybinette. 1997. Computing global virtual time in shared memory multiprocessors. *ACM Trans. on Modeling and Computer Simulation* 7, 4, 425–446.

R. M. Fujimoto, K. S. Perumalla, H. Wu, M. Ammar, and G. F. Riley. 2003. Large-scale network simulation: How big? how fast? In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 116–123.

F. Gomes, J. Cleary, and B. Unger. 1998. A survey of GVT algorithms. Technical Report pages.cpsc.ucalgary.ca/∼gomes/PAPERS/GVT.ps, University of Calgary, Canada.

A. O. Holder and C. D. Carothers. 2008. Analysis of time warp on a 32,768 processor IBM Blue Gene/L supercomputer. In *2008 Proceedings European Modeling and Simulation Symposium (EMSS)*.

D. R. Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 404–425.

D. R. Jefferson and H. Sowzral. 1982. Fast concurrent simulation using the time warp mechanism, Part I: Local control. Technical Report to the US Air Force N-1906-AF, Rand Corporation, CA, USA.

P. Konas and P.-C. Yew. 1992. Synchronous parallel discrete event simulation on shared memory multiprocessors. In *Workshop on Parallel and Distributed Simulation*. Vol. 24. SCS Simulation Series, 12–21.

Y.-B. Lin and E. Lazowska. 1990. Determining the global virtual time in a distributed simulation. In *International Conference on Parallel Processing III*. 201–209.

R. J. Lipton and D. W. Mizell. 1990. Time warp vs. Chandy-Misra: A worst-case comparison. In *SCS Multiconference on Distributed Simulation*. SCS Simulation Series, vol. 22. 137–143.

F. Mattern. 1993. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel Distributed Computing* 18, 423–434.

T. McLean, R. Fujimoto, and B. Fitzgibbons. 2004. Middleware for real-time distributed simulations. *Concurrency Computation: Practice and Experience* 16, 1483–1501.

D. Nicol. 1993a. Global synchronization for optimistic parallel discrete event simulation. In *Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, San Diego, CA, USA, 27–34.

D. M. Nicol. 1993b. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the Association for Computing Machinery* 40, 2, 304–333.

S. Oral, F. Wang, D. A. Dillow, R. Miller, G. M. Shipman, D. Maxwell, D. Henseler, J. Becklehimer, and J. Larkin. 2010. Reducing application runtime variability on jaguar xt5. In *Cray User Group Conference*.

A. J. Park, R. M. Fujimoto, and K. S. Perumalla. 2004. Conservative synchronization of large-scale network simulations. In *Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 153–161.

K. Perumalla, and R. Fujimoto. 2001. Virtual time synchronization over unreliable network transport. In *Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 129–136.

K. S. Perumalla. 2005. μsik: A micro-kernel for parallel/distributed simulation systems. In *Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 59–68.

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

K. S. Perumalla. 2010. $\mu\pi$: A scalable and transparent system for simulating MPI programs. In *3rd International Conference on SIMUTools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 62–67.

K. S. Perumalla and A. J. Park. 2011. Improving multi-million virtual rank execution in $\mu\pi$. In *Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 454–457.

K. S. Perumalla, A. J. Park, and V. Tipparaju. 2011. Gvt algorithms and discrete event dynamics on 129k+ processor cores. In *International Conference on High Performance Computing*. IEEE, 1–11 b.

K. S. Perumalla and S. K. Seal. 2012. Discrete event modeling and massively parallel execution of epidemic outbreak phenomena. *Transaction of Society for Modeling and Simulation International* 88, 7, 768–783.

B. Preiss, W. Loucks, I. MacIntyre, and J. Field. 1991. Null message cancellation in conservative distributed simulation. In *Advances in Parallel/Distributed Simulation*. Vol. 23. SCS Series, 33–38.

P. Reiher, F. Wieland, and P. Hontalas. 1990. Providing determinism in the time warp operating system -costs, benefits, and implications. In *Workshop on Experimental Distributed Systems*. IEEE, Huntsville, Alabama, 113–118.

Reynolds, J. Paul, C. Pancerella, and S. Srinivasan. 1993. Design and performance analysis of hardware support for parallel simulations. *Journal of Parallel and Distributed Computing* 18, 4, 435–453.

M. Rosu, K. Schwan, and R. M. Fujimoto. 1997. Supporting parallel applications on clusters of workstations. In *IEEE Symposium on High Performance Distributed Computing*, 159–168.

B. Samadi. 1985. Distributed simulation, algorithms and performance analysis. Ph.D. thesis, Department of Computer Science, University of California, Los Angeles.

D. Skinner and W. Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *IEEE International Workload Characterization Symposium*. 137–149.

Received February 2012; revised April 2013, September 2013, and January 2014; accepted March 2014