Devastator: A Scalable Parallel Discrete Event Simulation Framework for Modern C++

John Bachan
Tan Nguyen*
Mahesh Natarajan
Maximilian Bremer
Cy Chan
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Xuan Jiang
University of California, Berkeley
Berkeley, CA, USA

Jianlan Ye
Arizona State University
Tempe, AZ, USA

ABSTRACT

Parallel discrete event simulation is a fundamental simulation technology that is essential to the parallelization of event-based models including hardware and transportation systems. Parallelization is often difficult due to dynamic data-dependencies and limited computational work for hiding runtime overheads. We present Devastator, a scalable parallel discrete event simulation framework for modern C++. Devastator provides a productive API that leverages C++'s type system to eliminate boilerplate code. Devastator has been designed to specifically optimize performance on distributed many-core architectures with deepening memory hierarchies. Devastator relies on the GASNet-Ex communication runtime as well as lock-free message queues to achieve highly competitive performance. We perform strong and weak scaling studies on NERSC's Perlmutter up to 32,698 cores and demonstrate an up to 5x speed-up over the ROSS simulator for workloads with substantial locality.

CCS CONCEPTS

• **Computing methodologies** → **Discrete-event simulation**; *Modeling methodologies*; • **Applied computing** → *Transportation.*

KEYWORDS

High-performance computing, parallel discrete event simulation, Time Warp, active messages, GASNet

ACM Reference Format:
John Bachan, Jianlan Ye, Xuan Jiang, Tan Nguyen, Mahesh Natarajan, Maximilian Bremer, Cy Chan. 2024. Devastator: A Scalable Parallel Discrete Event Simulation Framework for Modern C++ . In *38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS '24), June 24–26, 2024, Atlanta, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3615979.3656061

*Corresponding author: Tan Nguyen, TanNguyen@lbl.gov, (510) 486-5518, 1 Cyclotron Road MS:59R4010A, Berkeley, CA 94720

SIGSIM PADS '24, June 24–26, 2024, Atlanta, GA, USA

ACM ISBN 979-8-4007-0363-8/24/06
https://doi.org/10.1145/3615979.3656061

1 INTRODUCTION

Discrete event simulation (DES) is a class of simulations that expresses a model as a series of timestamped events that occur on pre-defined actors and may schedule subsequent events based on local actor information. DES is a fundamental simulation technology, playing a key role in accelerating the simulation of vehicle transportation [8, 20, 26, 35], hardware simulation [11, 28, 31, 41, 43] as well as many other domains [38, 39, 44].

The key challenge for discrete event simulations is how to parallelize them. The design contract of a parallel discrete event simulation (PDES) engine promises that by the end of the simulation the local state modifications due to event execution are identical to that of the sequential model. Since event scheduling depends on local simulation state and predicting the arrival of events on an actor is not generally possible, parallelizations typically fall into two main categories. *Conservative* approaches leverage model specific information to guarantee that no events can arrive within a given time interval, indicating that work within a time interval is safe to execute. *Optimistic* synchronization does not require that all events will be executed in the correct order, but only requires that events that are executed in an incorrect order are replayed in a manner that allows the final actor state to be consistent with the correctly executed events.

Drawing generalizations about the performance trade-offs between the two parallelization schemes is difficult [18]. Conservative approaches commonly expose less parallelism to the simulator and require detailed knowledge about the problem to expose sufficient parallelism to obtain good performance. Optimistic approaches on the other hand suffer from complications when too many events have been incorrectly executed leading to substantial runtime overheads. Furthermore, the requirement that an event be "reversible" often comes with substantial coding overheads, even with automated approaches [33, 46, 51], precluding the coupling of third party libraries without substantial engineering effort.

PDES is a mature simulation technology with many of the seminal papers being written in the 1980s and 1990s. However, much of the work of that time was confronting a different architectural reality. Many old papers focus on not exceeding memory capacity constraints and focusing on the cost of computation. On today's supercomputers, the critical constraints couldn't be more different. While FLOPs have become readily available, simulation times are now constrained by the cost of synchronization, accessing memory

across deepening memory hierarchies, and the latency of distributed communication. Many PDES applications are either memory latency or bandwidth bound.

In that spirit, this paper presents a modern high performance implementation of Jefferson's TimeWarp [22] protocol. We introduce a hybrid parallelism framework where shared memory messages are sent using lock-free message queues. Inter-process and inter-NUMA domain messages are handled using the GASNet-EX runtime [5], which provides low level support for a variety of interconnects.

Another key barrier to adoption has remained the programmability and usability of PDES simulators. Many simulators such as ROSS [7] and Root-SIM [34] are written in C and require verbose specifications of actors and events. NS-3 [31] is a conservative framework and it requires the users to implement lookahead optimizations for more parallelism. Other PDES engines, such as Simian [45] are written in Python and provide substantially lower throughput. We aim to bridge this gap through the introduction of a modern C++ API. Using active messages, modern memory allocators, and efficient serialization routines, users can efficiently implement their discrete event models while still achieving performance that is competitive with simulators like ROSS. Differentiating software packages via programming languages is largely a matter of personal preference, but we believe the introduction of a parsimonious C++ PDES library with performance and scalability comparable to ROSS make our software solution a valuable contribution to the PDES community. The main contributions of this paper are as follows.

(1) We introduce Devastator, a high performance C++ optimistic Time Warp-based parallel discrete event simulator designed for distributed multi-core supercomputers. Devastator comes with numerous advanced features, including adaptive lookahead window, lock-free message queues and memory allocator, and other cache/memory aware optimizations. For better scaling on large systems, Devastator employs GASNet-EX communication runtime that can speedup data movement and facilitate object migration with active messages.
(2) We design an API that leverages modern C++ functionality to specify events and introduce heuristics to automatically tune the optimism of the simulator. The result is a programming model that is easier to develop and requires setting fewer parameters to run efficiently.
(3) We showcase Devastator's performance by demonstrating the scalability of a series of PHOLD [19] benchmarks with varying parameterizations. We showcase the scalability of the simulator on NERSC's Perlmutter achieving 30-80% of single node per-core throughput at 32,698 cores and compare the performance to the ROSS simulator for which we achieve an up to 5x speed-up for problems with substantial locality.
(4) We also evaluate Devastator with a transportation network simulation. Although this experiment runs at a small scale, the results demonstrate the benefit of the adaptive lookahead window on dynamic, fast changing input data.

The remainder of the paper proceeds as follows. We outline the important algorithmic features and the API in Sec. 2. In Sec. 3, we describe the communication runtime. In Sec. 4, we present Devastator's performance and compare it to ROSS, and in Sec. 5, we simulate the traffic in the Los Angeles metropolitan area and analyze the performance. Finally, we present the related work in Sec. 6 and conclude the paper in Sec. 7.

2 PARALLEL DISCRETE EVENT SIMULATION

Discrete event simulation (DES) is a computational paradigm for simulating a time-evolving system which is driven forward by instantaneous state changes at prescribed timestamps often referred to as *events* [9, 21, 24]. In addition to modifying the state, each event can also register new events to execute at future timestamps. The application is defined by its state space, initial state, and initial set of events. The sequential DES framework runtime is responsible for executing the events in timestamp order until no further events exist, at which point the simulation is complete. Parallel discrete event simulation (PDES) is an attempt to extract parallelism from this paradigm. In PDES, applications are additionally required to partition their state, with each partition referred to as a logical process (LP), such that any event only accesses state pertaining to its designated LP. This makes events on different LP's trivially safe to execute concurrently, but drastically complicates the runtime's task of ensuring events are executed in timestamp order on their respective LP.

There are two primary execution strategies for PDES simulators: conservative and optimistic. In conservative execution, the runtime only executes an event when it has proven that no other event could possibly be generated to precede it. Time Warp, an optimistic approach, first described by Jefferson [22], takes the opposite approach where it assumes that whatever event a CPU has in-hand, given that it has the least timestamp for its LP, is probably safe to execute despite the possibility of receiving an event from a peer CPU at some later time of a new event with lower timestamp. To handle this eventuality, the runtime must be capable of "rolling back" the execution of previously executed events to revert the LP back to the state when it ought to have executed the new event. This requires the application to supply the runtime with such a means to reverse execution, a non-trivial endeavor itself, and also be capable of executing in contexts generated by invalid event orderings (which will eventually be rolled back). The advantage this brings over conservative execution is that the runtime can execute events synchronization free and capitalize on the common occurrence of events being safe to execute despite lacking a proof of the fact.

2.1 Asynchronous GVT Computation

A key part of any Timewarp-based PDES simulator relates to the computation of global virtual time (GVT). Ensuring that a sequence of events is reversible requires the maintenance of information that can be used to restore the state to an arbitrary point in past simulation time. This can require a substantial amount of memory. Global Virtual Time (GVT) is a simulation time for which all events in the system have already been processed. Once GVT exceeds the timestamp of an executed event, these events' artifacts can be safely destroyed freeing up memory.

In light of Devastator's emphasis on performance and scalability, we opt for an asynchronous version of TimeWarp, which avoids the use of any blocking collectives. Our method is phase-based [27] and largely follows in the footsteps of [36, 37]. Each rank is persistently

performing GVT reductions in the background. Upon completion of a GVT reduction phase, we obtain a new bound on GVT, which triggers fossil collection, and immediately begin the next GVT reduction.

A key challenge in obtaining a high performance Timewarp implementation requires limiting of undue optimism. Since a rolled-back event must also roll back the events it has sent to neighboring processors, the runtime overhead can quickly overwhelm any benefits of the additional parallelism exposed through optimistic PDES execution. Numerous methods have been proposed to balance the overheads due to rolling back with the time spent in GVT reductions [12, 13, 17, 30, 42, 48–50]. Devastator adopts a variant of the moving time window (MTW) [47, 53]. The high-level idea is: given a GVT, only execute events within the time interval GVT to GVT+w, where w is a tunable parameter.

To eliminate the need for setting parameters, we use a hill climbing algorithm to compute the optimal window size. Devastator maintains a moving average of *globally* committed and executed events over the past 16 GVT reductions. We refer to the efficiency as the ratio of committed over executed events. The window size is adjusted after the completion of each GVT reduction as follows:

(1) If the efficiency is below 33%, the window size is quartered.
(2) If the efficiency is between 33% and 66%, the window size is halved.
(3) If the efficiency exceeds 95%, the window-size is doubled.
(4) Otherwise, the window size increases or decreases by 1%, based on if the number of committed events increased or decreased.

To ensure forward progress, the window size is at least 1. To avoid sending additional messages, these throughput metrics are piggybacked onto the GVT reductions.

2.2 PDES API

Devastator touts its C++14 based API for describing a PDES application as one of its major achievements due to it being succinct yet incredibly expressive. The program first defines a set of events and their behaviors.

2.2.1 Event specification. Events can be objects of any C++ class type so long as it exposes proper `execute` and `unexecute` methods.

Listing 1 presents an example of how a user might specify an event. This example `my_event` class sends a message along a chain of logical processes (LPs). Devastator supports multiple LPs per rank (ranks run to completion while LPs can be preempted), but for the sake of simplicity in this example we use one LP per rank. The user must specify one function for the `event` class. The `execute` member function executes the event. For this example, a value is hashed based on the local hash state. Devastator provides to these user events a `deva::pdes::execute_context` object. This object includes the local current LP ID (causality domain or `cd`) and the simulation time. Additionally, the execute context contains a member function '`send`', which allows the user to schedule future events. This member function performs the bookkeeping required for the GVT computation. The return argument of the execute function is another user-defined object that reverses the event. The return object will be passed into Devastator, which will maintain its lifetime until it can be either committed or unexecuted. This generic

```
thread_local uint32_t my_hash = 0; // LP state
struct my_event { // Event class
 uint32_t mixin; SERIALIZED_FIELDS(mixin)
 struct reverser {
   uint32_t prev_hash;//for reversing computation
   void unexecute(deva::pdes::event_context&,
        my_event &e) { my_hash = prev_hash;}
   void commit(
       deva::pdes::event_context&,
       my_event &e) {/*optional*/}
 };
 reverser execute(deva::pdes::execute_context &cxt) {
   uint32_t prev_hash = my_hash;
   my_hash ^= mixin; my_hash *= 0xbeef;
   // send an event elsewhere
   if(deva::rank_me()+1 != deva::rank_n) {
     cxt.send(/*rank*/deva::rank_me()+1, /*cd*/0,
       /*time*/cxt.time+1,my_event{mixin + 1});
   }
   return reverser{prev_hash};
 }
};
```

Listing 1: Event behaviors are defined by three functions: execute, unexecute, and commit. In this example, new events are created and sent along a chain of LPs.

approach allows the user to specify how events are rolled back. The unexecute member function of the reverser is called when an event is rolled-back and expected to restore the LP state to prior to the event execution. A commit function may be specified, which will be invoked prior to fossil collection. To enable sending events between processes, Devastator utilizes the UPC++ [2] serialization API to efficiently convert objects into buffers that can be deserialized by the receiving process. In this example, we use a simple `SERIALIZED_FIELDS` macro.

LP state is accessed by locally accessible global variables. Devastator provides no high level LP abstractions. This comes with advantages and disadvantages. Typically Devastator applications maintain thread local data structures that can be indexed into via causality domain IDs. This approach permits the user to specify how the local LP state should be laid out and stored. Drawbacks include a lack of automated rollback strategies, such as those based on tracking address spaces associated with LPs [33, 51]. Additionally, it requires the user to maintain data structures to map an LP ID to a rank-causality domain ID pair.

The primary differences between the Devastator API and existing ROOT-Sim and ROSS simulators revolve around utilization of C++'s support for more complex data types and type deduction capabilities. The outlined PDES API enables users to send messages with a single invocation of `execute_context::send`. The event types are automatically serialized and can contain arbitrary datatypes (vis-a-vis ROSS, which prohibits the sending of dynamically allocated data). Additionally, C++'s support of type deduction eliminates the need for registering different event types and handlers (as is the case in ROSS) or de-multiplexing event-types as part of

the `processEvent` callback in ROOT-Sim. Through use of a type-erasure idiom, Devastator can automatically build these look-up tables during compile time. Devastator's use of the C++ library eliminates boiler plate code and creates a more readable code.

2.2.2 Program Control Flow and Event Management. A small part of the PDES runtime is embedded in user-accessible outer layer resembling the typical SPMD distributed parallel model, which allows the user to invoke multiple PDES simulations per job run, or to temporarily break from PDES execution to engage in collective operations and then resuming PDES. Listing 2 shows the API. Before calling into the PDES layer, each process will initialize the communication runtime via a call to `deva::run` (Section 3). Thereafter, each worker thread must specify the number of local LPs as the argument to the `deva::pdes::init`. The user schedules initial events using `deva::pdes::root_event`. When ready, `deva::pdes::drain` executes all events up until virtual time `t_end`. Invocations of `deva::pdes::drain` provide the ability to globally pause the simulation at virtual timestamps.

Another feature of drain invocations is that by setting the *reversible* argument to true will not perform any fossil collection. A collective invocation to `deva::pdes::rewind` will restore the simulation state to prior to the previous drain invocation. Repeated calls to drain and rewind allow users to explore control scenarios or run model systems with uncertain parameters. Lastly, the `deva::pdes::finalize` function cleans up the data structures.

3 COMMUNICATION RUNTIME

Devastator is composed of numerous software components, including event scheduler, worker threads and a distributed communication system. Here we describe the design and implementation details of the communication component that differ significantly from common runtime systems. We also discuss domain knowledge that is used for performance optimizations.

3.1 Communication Backend

The general structure of the communication engine follows the SPMD paradigm prevalent in HPC. It also exposes a productive and high-performance means to send Active Messages using modern C++ syntax, as well as a few other global collectives for productivity. Like MPI, we refer to our parallel execution agents as "ranks", but unlike MPI where those ranks correspond to OS processes, ours map to OS threads with the intention of there being many per process. Since an active message can be sent between any two ranks, this implies we support global thread-to-thread messaging, whether those threads belong in the same process or not.

The fact that we endorse messaging within an address space (between threads of the same process) is atypical within mainstream communication models. Messaging constructs are usually reserved for agents that do not share an address space, since it is often assumed that some shared memory mechanism can better service the need with lower latency. But the drawback to shared state is the on-demand migration of cache lines by the hardware, and the cost of this presents itself as CPU stalls on L1 cache misses. When the amount of data being "mixed-in" to the shared state is sufficiently smaller than the required memory traffic to perform that mixing, it can become advantageous to switch to a strategy where the state is

```
namespace deva {
namespace pdes {
  // Collectively initialize pdes engine
  // given number of CD's resident to
  // this rank. CD's (causality domains)
  // are the API's nomenclature for LP's.
  void init(int32_t cds_this_rank);
  // Submit a root event for a local CD.
  template<typename Event>
  void root_event(int32_t cd,
              uint64_t time, Event &&e);
  // Collectively process all events
  // with time < t_end.
  void drain(uint64_t t_end = UINT64_MAX,
             bool reversible = false);
  // Revert state via unexecute's as
  // it was upon entering last drain.
  void rewind(bool do_rewind);
  void finalize();// Tear down pdes engine
  struct event_context {
    int32_t cd;
    uint64_t time, subtime;
  };
  struct execute_context: event_context {
    template<typename Event>
    void send(int32_t rank, int32_t cd,
              uint64_t time, Event &&e);
  };
}}
```

Listing 2: Devastator employs the user-friendly SPMD model for program construction (init), initialization (`root_event`), the transition to the execution of a time window (drain and rewind), and program tear down (finalize). Events within a time window run asynchronously. An Event can be scheduled as soon as all data and resource constraints are met. Also, events' operations (e.g. send) do not block other events.

partitioned across processing cores, and messages are sent to the owner of the state to perform the mixing on the sender's behalf. We believe that PDES is an excellent candidate for this approach for two reasons. First, that events (the messages) tend to be on the order of just a few cache lines, while the shared state is a sorted queue of events that would be costly to migrate in the memory hierarchy, and second, inserting an event in the queue is a no-result operation, so the sender need not stall for its completion which permits the latency of processing the message at the target to be hidden.

3.2 Communication Flow

Whenever an event issues a communication request (e.g. `cxt.send()` in Listing 1), the rank's communication thread handles that request. The primary mechanism for ranks to communicate is through the `deva::send(Fn,Arg...)` routine, which asynchronously ships a function and arguments (the active message) to another rank where it is invoked at an arbitrary time during some invocation of

`deva::progress()`. The Devastator runtime has been designed to optimize the case where sends are numerous and small in size (< 1KB each) as this is the primary communication operation needed during a typical PDES simulation.

Devastator also provides a small set of collective operations akin to those of MPI, but are not at this time optimized for performance beyond using a textbook scalable algorithm. These are available to the application between PDES invocations, for example to gather statistics, IO, or compute load balancing. The essential prototypes of the Devastator communication API are presented in Listing 3.

```
namespace deva {
  // Initialize runtime, spawn threads
  // invoke `rank_main` on each thread
  template<typename Fn>
  void run(Fn rank_main);
  // Poll for incoming/outgoing communication
  void progress();
  // Send function & arguments to be
  // invoked during progress() at rank
  template<typename Fn, typename ...Arg>
  void send(int rank, Fn &&fn, Arg &&...arg);
  void barrier(); // Typical SPMD collectives
  template<typename T, typename Op>
  T reduce(T val, Op op);
  template<typename T, typename Op>
  std::pair<T/*prefix*/, T/*total*/>
    scan_reduce(T val, T zero, Op op);

  // More functions exist for detecting
  // the process/thread topology to
  // facilitate the application using
  // shared memory communication
  // protocols when advantageous...
}
```

Listing 3: We employ GASNet for a low latency, scalable communication runtime. Non-collective routines (e.g. send) do not block the computation. They can also include remote work executing at the recipient's address space (active message). We dedicate a processor core to handle the communication routines and regularly call progress() to ensure that active messages can be executed in a timely manner. Collective routines such as barrier() and reduce() are useful for coarse grain synchronization, but they block all the ranks and should not be called often

3.3 Configuration and Optimization

The Devastator runtime can be deployed in two different regimes of parallelism: single-process multi-threaded (non-distributed) and multi-process multi-threaded (distributed). In non-distributed runs, all threads belong to the same process and can all share the same address space. The messages sent between threads are not serialized, instead C++ move semantics are used to migrate the object into the target thread's message queue allowing for the greatest degree of copy elision. In distributed runs, address spaces are per-process and so require the application to be cognizant of the process/thread hierarchy to share memory meaningfully. Messages to process-local threads are shipped with `std::move` as before, but off-process messages must be serialized, transmitted, and deserialized on the target side.

Devastator has several message queue implementations for the thread to thread messaging service that can be selected at build time, enabling the user to make the optimal choice. Both are lock-free implementations. We support all-to-all single-producer single-consumer (SPSC) queues where a dedicated queue exists between every pair of threads in a process and is excellent for low to moderate thread counts, and all-to-one multi-producer single-consumer (MPSC) queues which are better suited to the higher thread counts of the many-core era due to the non-scalability of N^2 SPSC queues. To reduce latency handling thread contention, we use atomic read-modify-write operations instead of spin-locks. To further improve the scalability of the thread messaging service, we design a custom memory allocator called *deva malloc*. Like generic designs such as jemalloc, Deva's opnew malloc uses thread private size segregated bins for small objects. Deva malloc scans the private bins periodically to send memory back to the thread which originally allocated it using the same thread messaging queues we used for everything else. This design allows deva's opnew to stay lock free almost everywhere which is fundamentally not possible in jemalloc due to it not owning the lifetime of threads.

For multi-process builds, our inter-process messaging capability is provided by the GASNet library, a portable and high performance layer for HPC communication. While MPI may be the de-facto standard for message passing in HPC, our heavy reliance on active messages makes MPI unattractive in this context. GASNet has supported active messages since its inception and has a highly optimized implementation for each of the major HPC networks it supports. Regarding the communication mode, we had two options: i) threaded mode where all worker threads can safely invoke communication calls and ii) single communication thread handles all communication requests from its worker threads. The former incurs a significant cost as it forces the communication runtime to use synchronized access to the underlying fabric. For this reason we have chosen to configure GASNet in single communication thread mode, and we dedicate an internal "service" thread to perform all GASNet operations. The service thread enjoys not only synchronization-free access to the network, but GASNet's internal state needed to conduct network operations also enjoys greater locality since it runs on a fixed CPU that does little else. This is exemplary of our tenet to partition state and send messages rather than synchronizing to share state. The downside though is that a message must now experience a three hop latency to get to its destination: an initial thread-to-thread message to the local service thread, a GASNet active message to the destination process, and then a final thread-to-thread message from the remote service thread to the destination thread. This cost seems worth it, especially in light of the added optimization where the service thread greedily agglomerates outgoing messages sharing the same destination process to increase average message size and amortize the impact of per-message overheads.

4 PHOLD BENCHMARK

4.1 Benchmark setup

To showcase the performance of Devastator for arbitrarily sized problems, we employed a modified PHOLD benchmark [19]. In PHOLD, each LP is initialized with events and each of which will only schedule a subsequent future event on the current LP or one of its peer LPs. Thus, each chain of events is analogously described as a ray reflecting among LPs, so the workload can be measured with the number of rays. Message delivery times are delayed by a value sampled from an exponential distribution. The destination LP of the following event is obtained by sampling from a Gaussian distribution centered at the sending LP. The standard deviation (σ) is adjusted to control the fraction inter-node and intra-node communications. We opt to introduce this alternative reflection mode as we find a uniform transition probability of the original PHOLD bears little resemblance to Devastator's real world workloads, but which instead contain substantial locality.

4.2 Hardware configurations and runtime environments

We utilize the CPU partition of NERSC's Perlmutter and TACC's Frontera machine for these performance studies. On the Perlmutter system, each compute node has 2 sockets and each socket has AMD EPYC 7763 CPUs, providing a combined total of 128 physical cores. The nodes are connected with Hewlett Packard Enterprise Slingshot 11 interconnect technology. On Frontera, each compute node contains 2 28-core Intel 8280 "Cascade Lake" processors. The nodes are connected via a Mellanox Infiniband, HDR-100 interconnect. The runtime environment information is tabulated in Table 1.

	Perlmutter	Frontera
Compiler	GCC-Cray v11.2.0	GCC v9.1.0
GASNet Version	2023.3.3	2023.3.3
GASNet Conduit	ofi	ibv
Conduit Version	libfabric v1.15.2.0	ibverbs v20.35.2000

Table 1: Compiler and runtime versions

4.3 Determining the optimal configuration

We first conduct a parameter sweep to find the optimal configuration to run the Devastator on Perlmutter and Frontera. Devastator contains numerous parameters to configure. We compute throughput on a matrix of parameters including number of processes per node, the three Devastator-supported memory allocators—namely *deva*, *libc*, and *jemalloc* [16], where the *deva* is a novel allocator inspired by the *jemalloc*—,and the two lock-free message queue implementations, SPSC and MPSC.

To focus on the behavior of local communications, we use the Gaussian transition probabilities with a small value of 100 for σ, and assign 100 LPs to each core. Four nodes are used in each Perlmutter run and two nodes are used in each Frontera run. This requires inter-node communication during the calibration. To investigate the impact of the problem size, we tested 400,000 rays and 800,000 rays for each configuration. The result is presented in Figure 1.

First of all, we do not observe significant impact of problem size on throughput for the optimal configurations on either architecture (less than 5%). For the memory allocators, *jemalloc* provides comparable throughput to *deva* on Perlmutter. However, on Frontera with 2 processes/node, the *deva* allocator is 1.4x faster than *jemalloc*. As we scale out on Perlmutter, *libc* provides increasingly competitive performance as the number of threads contending for memory resources decreases. Across both systems, we observe a substantial performance benefit associated with proper treatment of NUMA domains. Comparing the 800k rays/*deva* configurations for 1 process per node to 2 processes/node on Frontera and 8 processes/node on Perlmutter, we observe speed-ups of 2.3x and 8.1x, respectively. This emphasizes the importance of properly accounting for NUMA effects on modern multi-core systems, which has been also reported in recent PDES work [40]. On Perlmutter, we note that the performance for 4 and 8 processes/node provides the best and most consistent performance. Since the Perlmutter node only has two sockets, the performance discrepancy between the SPSC and MPSC nodes is somewhat of a surprise, but probably due to the lower efficiency of the SPSC runs. For the remainder of the section, we use the *deva* memory allocator, SPSC message queues and 2 processes per node on Frontera, and MPSC message queues and 8 processes per node on Perlmutter.

4.4 Weak and strong scaling

To better understand how the Devastator performance scales in massive parallel runs, we conducted weak and strong scaling tests with the Gaussian reflection modes with different standard deviation σ. We selected four σ values to control the degree of communications locality, namely σ = 100, 6400, 10000, and 25600. For example, when σ = 100, about 60% of the messages were designated to route to other cores and almost no message was sent to other nodes, whereas when σ = 10000, almost all messages (99.6%) were sent to the other cores and 55% the messages sent to other nodes.

Fifteen rays are assigned to each LP and 100 LPs are assigned to each core in the weak scaling test; we therefore have 192K rays for each node. The simulation ends after each ray executes 50K times. In the strong scaling test, 400K rays are used regardless of the number of nodes used. Three runs are conducted in each configuration to estimate the noise level. The data is presented in Figure 2.

In the weak scaling, we scaled the simulation up to 256 nodes (32768 cores), maintaining a per-core commit rate of about 30% to 80% of the single node per-core commit rate. Although the data appears to be noisy, the overall decreasing trend of parallel efficiency remains valid over the range of the number of nodes. The efficiencies of the runs, represented by the ratio of committed events and executed events, were also monitored and presented in Figure 2(e); where it can be observed that the efficiency is well controlled in the range of 60% to 80% with our dynamic lookahead algorithm.

In the strong scaling, we were also able to achieve a parallel efficiency of about 50% on 16K cores, corresponding to speed-ups of 56x-70x. It is worth noting that the parallel efficiency is larger than one at two nodes for all the values of σ, caused by bad single-node performance and can be attributed to two factors. Firstly, when all the workloads are concentrated on a single node, the cache could be overflowed leading to more cache misses. Secondly, all the

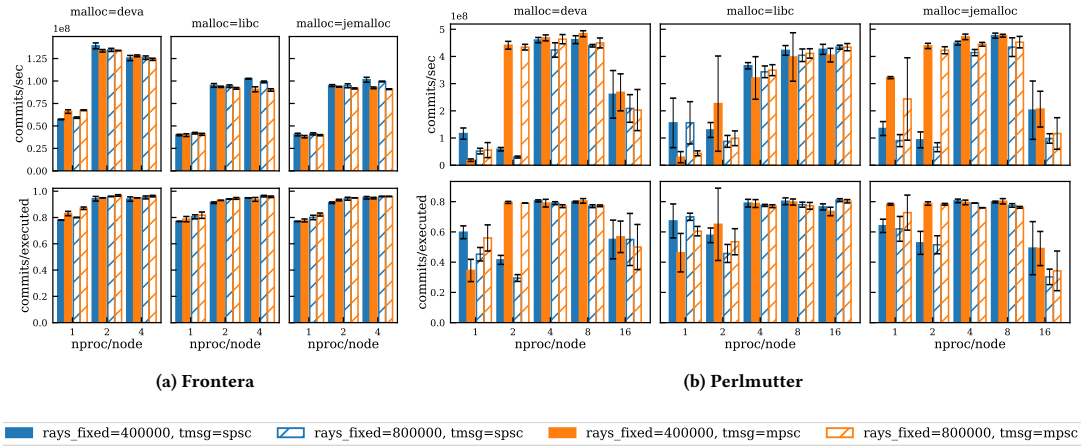

Figure 1: The performance using different memory allocators, message queues, and number of processes. Three columns of subplots include the data with different memory allocators as demonstrated on the top of the columns.

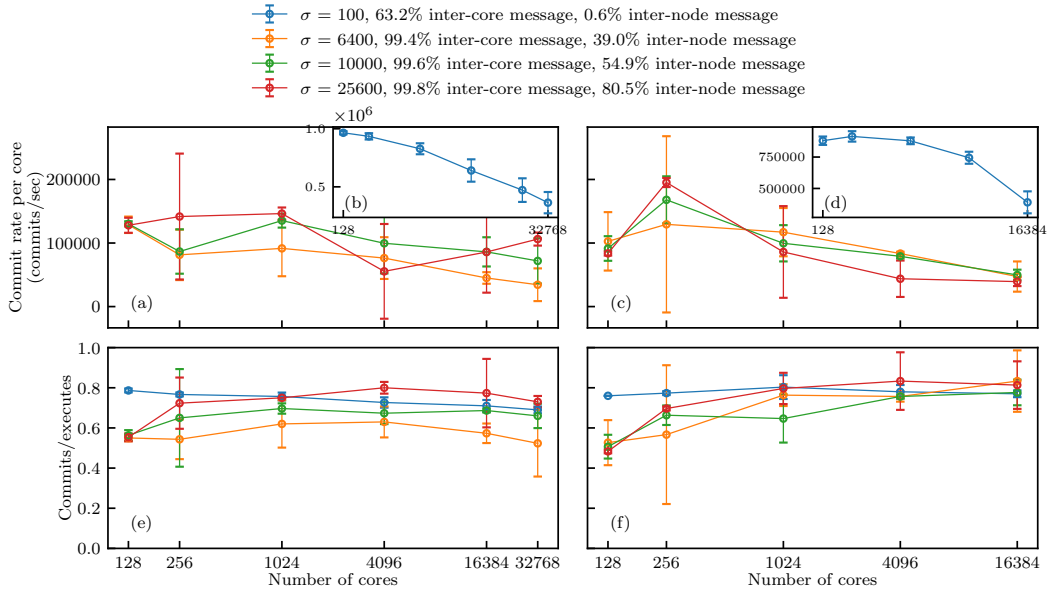

Figure 2: The weak scaling results are shown in subplots (a), (b), and (e) on the left column, and strong scaling results are shown in subplots (c), (d), and (f) on the right column. The σ represents the standard deviation of the Gaussian distributions that determine the target LPs.

communication, either designated to be inter-node or intra-node, are actually intra-node communications in single-node simulations and could overwhelm the memory bandwidth of the node. An observation can serve as evidence of the overwhelming memory access: when transitioning from one node to two, a larger increase in the per-core commit rates is observed when there are more designated inter-node messages because the designated inter-node messages were delivered to the other node, mitigating the demand for intra-node memory access and thus improving the performance.

4.5 Comparison of Devastator and ROSS

We conclude our study of the PHOLD benchmark by comparing the performance of Devastator to that of ROSS [7]. For a fair comparison, we build a common API such that both simulators use the identical application code. Due to challenges with Perlmutter's vendor MPI implementation, we performed the comparison on TACC's Frontera machine. As the benchmark, we use the Gaussian PHOLD benchmark with standard deviations of $\sigma = 100$ and $\sigma = 10000$

to emphasize the impact of Devastator's hybrid parallelism. The commits per second are shown in Fig. 3.

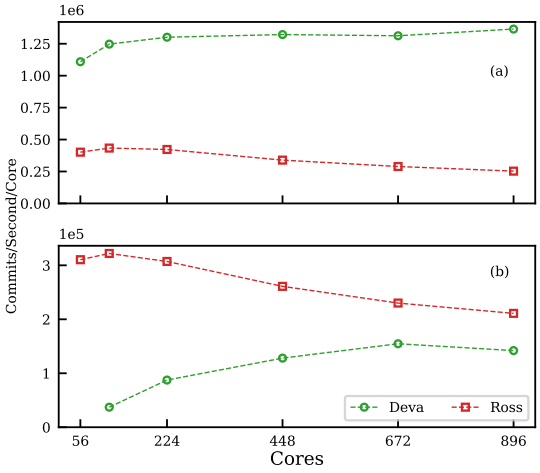

Figure 3: Devastator versus ROSS on Frontera for the Gaussian PHOLD benchmark with (a) $\sigma = 100$ and (b) $\sigma = 10^4$.

The comparison of ROSS to Devastator shows strong dependence on the workload. For $\sigma = 100$, which corresponds to substantial locality in the event workloads, Devastator outperforms ROSS by 2.9x (2 nodes) to 5.4x (16 nodes). When the standard deviation increases, the ROSS outperforms Devastator by 8.7x (2 nodes) to 1.5x (16 nodes). We believe that the poor Devastator performance arises from the problem size. With so many messages flying around, the asynchronous GVT algorithm is slower to complete and the adaptive moving time window is slow to adapt. Addressing this issue is a topic of future work.

Lastly, we note that we used a single Devastator configuration for all runs. On the other hand, each ROSS configuration at each node count required a grid sweep of 45 runs per node to determine the optimal `batch` and `interval` parameters. Thus, even in cases where ROSS was substantially faster, ROSS required substantial tuning, reducing user productivity.

5 APPLICATION

The evaluation of our implementation of Devastator in Mobiliti [8] aims to answer two research questions:

- How do Devastator's main algorithmic features contribute to its performance?
- Can the proposed adaptive MTW provide substantial speedup for simulations of traffic propagation networks?

5.1 Benchmark setup

Mobiliti [8] is a large-scale, parallel discrete event simulation designed for analyzing transportation systems built based on Devastator, which is the first time that PDES is applied in transportation. In Mobiliti, vehicles are assigned routes one of two ways. In Dynamic Traffic Assignment (DTA), an optimization of a utility function is performed to generate a static set of routes for all the vehicles. Alternatively, dynamic rerouting routes vehicles using a contraction hierarchy, but updates the routes in response to evolving congestion patterns. Mobiliti is run with 4 processes per node on Perlmutter.

Our analysis of Los Angeles' transportation and urban landscape uses the Southern California Association of Governments (SCAG) travel demand model [14], providing a thorough understanding of regional travel patterns. For the road network, we have integrated geospatial data from HERE map [1], ensuring that our findings are accurate and up-to-date. By combining these diverse but complementary data sources, our paper presents a nuanced and detailed depiction of the urban and transportation intricacies of Los Angeles. The LA model is substantially larger than previously reported Mobiliti simulations. Where previous SF runs consisted of 19.2 million trips on a network with 450 thousand nodes and 1 million links, the LA model schedules 49.2 million trips on a road network of 2 million links and 903 thousand nodes.

5.2 Mobiliti Scalability

Mobiliti is parallelized by partitioning the road network across ranks. Historical traffic runs are used to weight links before being partitioned by METIS [23]. Load balance achieved may deteriorate during the day as traffic patterns deviate from an average traffic pattern, leading to congestion and irregularities in flow distribution at the end of the day.

In Figure 4, we showcase the parallel speed-ups of Mobiliti. The dynamic rerouting scenario exhibits runtimes from 1.2 million seconds to 181,365 seconds, corresponding to parallel efficiency of 88%. The DTA case, which contains less compute work, runs on a single node in 452 seconds and achieves a best time to solution of 179 seconds, corresponding to a parallel efficiency of 32%.

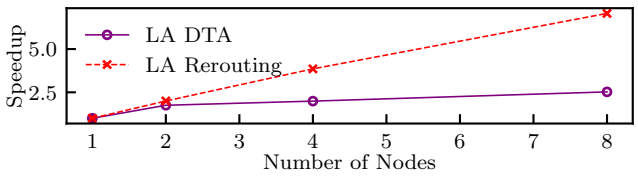

Figure 4: Parallel speed-up of Mobiliti Perlmutter relative to a single node (128 cores).

To understand the degradation in performance, in Figure 5, the execution time is broken down by the work being done. We observe a discernible uptick in the total execution cost associated with the spin look, which indicates the local event queues have events that cannot be executed due to the MTW throttle. This phenomenon becomes increasingly evident as we scale out to more nodes. The implementation of dynamic rerouting significantly contributes to a reduction in the overall execution cost. Recomputing the routes for all vehicles can be done in an embarrassingly parallel manner. However, it is important to note the presence of sequential components within the system, particularly the contraction hierarchy, which is recalculated by every thread. This unavoidable sequential bottleneck underscores the necessity of maintaining a delicate balance between parallel and sequential processing to optimize overall system performance.

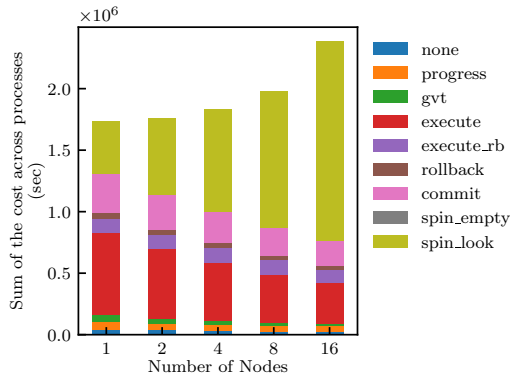

Figure 5: Timing Breakdown for Mobiliti with Rerouting.

In Figure 6, we define the imbalance metric, $\mathcal{I}$, as:

$$\mathcal{I} = \frac{\max(\tau) - \text{avg}(\tau)}{\text{avg}(\tau)}$$

where τ represents the execution time, i.e. time spent executing eventually-committed events. It is evident from the figure that $\mathcal{I}$ exhibits an increasing trend towards the culmination of the simulation. We believe given the slow worsening of load imbalance, semi-static approaches, which periodically pause the simulation to redistributed LPs such as those in [6] can be sufficient and plan to explore these options in future work.

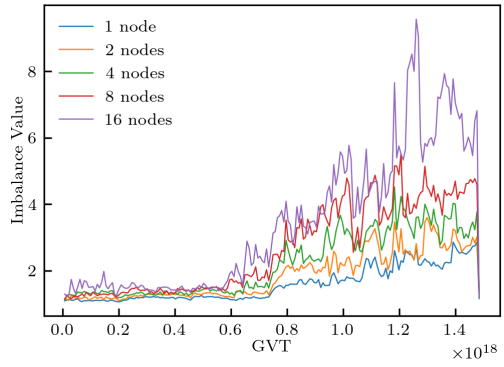

Figure 6: Compute imbalance versus GVT

We aim to confirm that the implementation of dynamic lookahead in GVT processing will enhance speed while maintaining consistent efficiency levels. Additionally, we seek to understand the implications of not implementing MTV. To achieve this, we conducted tests on Mobiliti both with and without dynamic lookahead. In Figure 7, we analyze the efficiency of Mobiliti with rerouting. Comparing a single node performance to four nodes, there is a subtle decrease in efficiency. However, it is paramount to highlight the high level of efficiency maintained across all nodes, underlining the robustness of the rerouting mechanism. Figure 8 shows that without dynamic lookahead, efficiency is decreased, and the overall runtime for the whole simulation is longer than with dynamic lookahead. For one node simulation, the runtime is 1641.9 seconds vs 1404.0 seconds, i.e. 1.17 times faster. For four nodes simulation, we get 428.1 seconds vs 369.0 seconds, i.e. 1.16 times faster.

6 RELATED WORK

Global virtual time (GVT) is at the heart of optimistic synchronization. By establishing a simulation time point before which all messages have been processed, memory resources for rolling back can be reclaimed. Synchronous GVT reductions execute events before entering a blocking call. The number of events is commonly limited by either event counts or virtual time increments. Typically, these methods require understanding the trade-offs between overheads due to roll back and time spent in GVT reductions.

Numerous strategies exist to limit undue optimism in parallel discrete event simulations [12]. Design of throttling methods typically vary based on how overly eager LPs are identified and how these LPs are delayed. Once eager optimism has been identified approaches for throttling LPs include stalling or suspending processes [49], modifying iteration counts in the main Time Warp simulation loop [50], probabilistically skipping event executions [17], limiting memory usage [13], customizing LP scheduling algorithms [30, 48, 57], forcing synchronization [15], load rebalancing [42], and leveraging domain specific information [4, 6, 25].

The approach utilized in Devastator is an adaptive formulation of the moving time window [47]. An adaptive formulation for window size based on "useful work"—the number of committed events per wall clock time—is formulated in [29], but they consider local windows rather than a single global window. In [54], they adaptively size the time window using reinforcement learning and demonstrate a 15% speed-up on a 4 core machine using VLSI simulation. A genetic algorithm approach is utilized in [55]. This allows for each LP to have a local time-window but requires running multiple concurrent simulations to tune parameters.

Computing systems have dramatically changed over the past 40 years, causing dramatic changes in how PDES simulators are best optimized for modern systems. Back in the early days, limited parallelism was exploited. DEVS is a formalism to describe discrete event systems [58]. It employs hierarchical decomposition of the model, allowing a model to be constructed from multiple sub-models. PDEVS further relaxes the serial constraints in DEVS for more concurrencies [10]. Fast forward to today, large scale PDES simulation [3, 36, 37] scale to hundreds of thousands or millions of cores. These approaches leverage a flat MPI model programming model. ROOT-Sim [34] has developed an optimized shared memory GVT reduction algorithm [32], which has been extended to distributed memory systems [52]. ROOT-Sim [34] performs dynamic memory allocation and release via the standard malloc library hooked by the kernel and redirected to a wrapper. The simulation platform has an internal state which distinguishes whether the current execution flow belongs to the application-level code or the platform's internals. In the former case, the hooked calls are redirected via the wrapper to an internal Memory Map Manager (called DyMeLoR), which handles allocation/deallocation operations by maximizing memory locality for the state layout for each single simulation object, and by maintaining meta-data allowing the memory map to be recoverable to past values. In addition, application

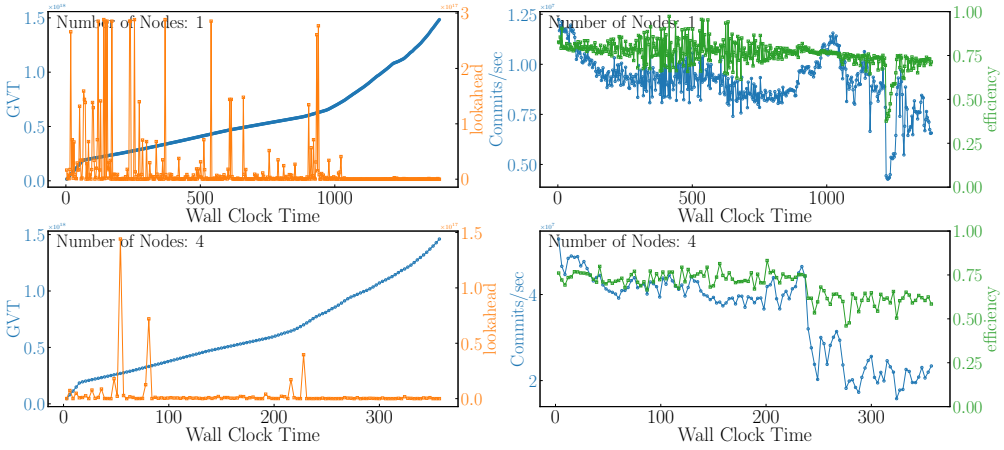

Figure 7: Progress with dynamic lookahead

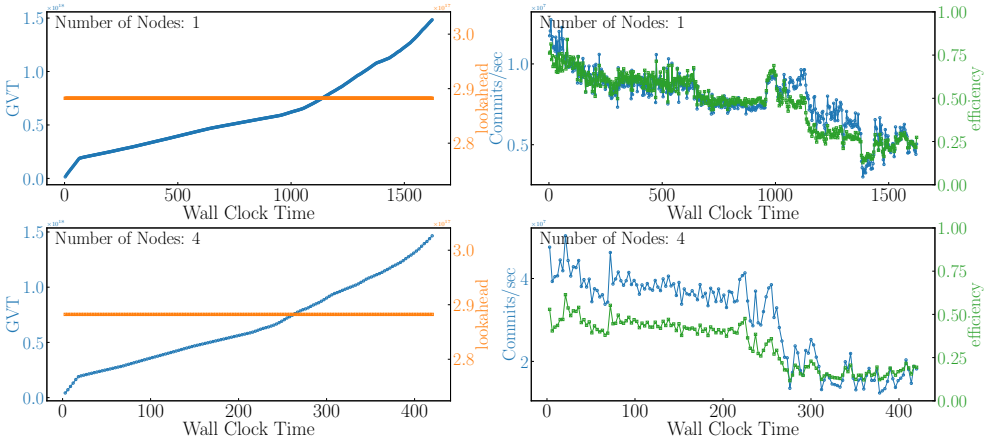

Figure 8: Progress with static lookahead

level software can be compiled via an ad-hoc light instrumentation tool so as to transparently provide the Memory Map Manager with the ability to track at runtime what memory areas are modified. In [56], ROSS-MC-CMT presented up to 10x speed-ups over ROSS (which uses priority queue algorithms such as calendar and heap [7]) by introducing shared memory message queues and introducing dedicated communication threads. Results are only presented out to 8 nodes. They also introduce an efficiency-based heuristic to tune the GVT interval size (i.e. the number of events executed per GVT reduction). They report no success using the moving average heuristic presented here. We believe this to arise due to the fact that Devastator utilizes an asynchronous GVT algorithm that is constantly performing reductions.

7 CONCLUSION

We presented Devastator, a software framework for optimistic PDES simulations. Devastator includes a new implementation of the Jefferson's TimeWarp algorithm so that PDES simulations can perform well on modern system architectures with deep memory hierarchy and a complex interconnection network. The C++ API also facilitates code development and maintenance process. At the system level, Devastator offers a few configuration and optimization options for the message queue and inter-node communication. Experimental results on up to 32K cores demonstrate performance and scalability benefits a program developed on Devastator can achieve compared to existing frameworks. For the next steps, we plan to develop extra hardware backends including GPUs and offer domain specific libraries for better programming productivity.

ACKNOWLEDGMENTS

This research was supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231, and used resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

REFERENCES

[1] ca. 2023. HERE Directions 2023: Uncover the latest innovations and developments in location technology. https://www.here.com/why-here. Accessed: 2023-10-30.

[2] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. 2019. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 963–973. https://doi.org/10.1109/IPDPS.2019.00104

[3] Peter D. Barnes, Jr., Christopher D. Carothers, David R. Jefferson, and Justin M. LaPre. 2013. Warp Speed: Executing Time Warp on 1,966,080 Cores. In *Conference on Principles of Advanced Discrete Simulation* (Montr©al, Québec, Canada). 327–336. https://doi.org/10.1145/2486092.2486134

[4] Pavol Bauer, Jonatan Lindén, Stefan Engblom, and Bengt Jonsson. 2015. Efficient Inter-Process Synchronization for Parallel Discrete Event Simulation on Multicores. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (London, United Kingdom) *(SIGSIM PADS '15)*. Association for Computing Machinery, New York, NY, USA, 183–194. https://doi.org/10.1145/2769458.2769476

[5] Dan Bonachea and Paul H. Hargrove. 2018. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18) (Lecture Notes in Computer Science, Vol. 11882)*. Springer International Publishing. https://doi.org/10.25344/S4QP4W https://doi.org/10.25344/S4QP4W.

[6] Maximilian Bremer, John Bachan, Cy Chan, and Clint Dawson. 2021. Speculative Parallel Execution for Local Timestepping. In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Virtual Event, USA) *(SIGSIM-PADS '21)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/3437959.3459257

[7] Christopher D. Carothers, David Bauer, and Shawn Pearce. 2002. ROSS: A high-performance, low-memory, modular Time Warp system. *J. Parallel and Distrib. Comput.* 62, 11 (2002), 1648–1669. https://doi.org/10.1016/S0743-7315(02)00004-7

[8] Cy Chan, Bin Wang, John Bachan, and Jane Macfarlane. 2018. Mobiliti: Scalable transportation simulation using high-performance parallel computing. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 634–641.

[9] K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (feb 1985), 63–75. https://doi.org/10.1145/214451.214456

[10] A.C.H. Chow and B.P. Zeigler. 1994. Parallel DEVS: a parallel, hierarchical, modular modeling formalism. In *Proceedings of Winter Simulation Conference*. 716–722. https://doi.org/10.1109/WSC.1994.717419

[11] Jason Cope, Ning Liu, Sam Lang, Phil Carns, Chris Carothers, and Robert Ross. 2011. Codes: Enabling co-design of multilayer exascale storage architectures. In *Proceedings of the Workshop on Emerging Supercomputing Technologies*, Vol. 2011. ACM.

[12] S. R. Das. 2000. Adaptive Protocols for Parallel Discrete Event Simulation. *The Journal of the Operational Research Society* 51, 4 (2000), 385–394. http://www.jstor.org/stable/254165

[13] Samir R. Das and Richard M. Fujimoto. 1997. Adaptive Memory Management and Optimism Control in Time Warp. *ACM Trans. Model. Comput. Simul.* 7, 2 (apr 1997), 239–271. https://doi.org/10.1145/249204.249207

[14] Richard G Dowling and Alexander Skabardonis. 2008. Urban Arterial Speed–Flow Equations for Travel Demand Models. In *Transportation Research Board Conference Proceedings*, Vol. 2.

[15] Ali Eker, Barry Williams, Kenneth Chiu, and Dmitry Ponomarev. 2019. Controlled Asynchronous GVT: Accelerating Parallel Discrete Event Simulation on Many-Core Clusters. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP '19)*. Association for Computing Machinery, New York, NY, USA, Article 64, 10 pages. https://doi.org/10.1145/3337821.3337927

[16] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[17] Alois Ferscha. 1995. Probabilistic Adaptive Direct Optimism Control in Time Warp. *SIGSIM Simul. Dig.* 25, 1 (jul 1995), 120–129. https://doi.org/10.1145/214283.214320

[18] Richard M. Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53. https://doi.org/10.1145/84537.84545

[19] Richard M Fujimoto. 1990. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulations, 1990*, Vol. 22. 23–28.

[20] Andreas Horni, Kai Nagel, and Kay Axhausen (Eds.). 2016. *Multi-Agent Transport Simulation MATSim*. Ubiquity Press, London. 618 pages. https://doi.org/10.5334/baw

[21] David Jefferson and Henry A. Sowizral. 1982. *Fast Concurrent Simulation Using the Time Warp Mechanism: Part I, Local Control*. RAND Corporation, Santa Monica, CA.

[22] David R. Jefferson. 1985. Virtual Time. *ACM Trans. Program. Lang. Syst.* 7, 3 (July 1985), 404–425. https://doi.org/10.1145/3916.3988

[23] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997 arXiv:https://doi.org/10.1137/S1064827595287997

[24] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[25] Jonatan Lindén, Pavol Bauer, Stefan Engblom, and Bengt Jonsson. 2017. Exposing Inter-Process Information for Efficient Parallel Discrete Event Simulation of Spatial Stochastic Systems. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Singapore, Republic of Singapore) *(SIGSIM-PADS '17)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/3064911.3064916

[26] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wiessner. 2018. Microscopic Traffic Simulation using SUMO. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. 2575–2582. https://doi.org/10.1109/ITSC.2018.8569938

[27] F. Mattern. 1993. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *J. Parallel and Distrib. Comput.* 18, 4 (1993), 423–434. https://doi.org/10.1006/jpdc.1993.1075

[28] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416635

[29] A.C. Palaniswamy and P.A. Wilsey. 1993. Adaptive bounded time windows in an optimistically synchronized simulator. In *[1993] Proceedings Third Great Lakes Symposium on VLSI-Design Automation of High Performance VLSI Systems*. 114–118. https://doi.org/10.1109/GLSV.1993.224467

[30] A.C. Palaniswamy and P.A. Wilsey. 1994. Scheduling Time Warp processes using adaptive control techniques. In *Proceedings of Winter Simulation Conference*. 731–738. https://doi.org/10.1109/WSC.1994.717422

[31] Joshua Pelkey and George Riley. 2011. Distributed Simulation with MPI in Ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques* (Barcelona, Spain) *(SIMUTools '11)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 410–414.

[32] Alessandro Pellegrini and Francesco Quaglia. 2014. Wait-Free Global Virtual Time Computation in Shared Memory TimeWarp Systems. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. 9–16. https://doi.org/10.1109/SBAC-PAD.2014.38

[33] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2009. Di-DyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects. In *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*. 45–53. https://doi.org/10.1109/PADS.2009.24

[34] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2012. The rome optimistic simulator: Core internals and programming model. In *4th International ICST Conference on Simulation Tools and Techniques*.

[35] Kalyan Perumalla. 2006. A Systems Approach to Scalable Transportation Network Modeling. In *Winter Simulation Conference*.

[36] Kalyan S. Perumalla, Alfred J. Park, and Vinod Tipparaju. 2011. GVT algorithms and discrete event dynamics on 129K+ processor cores. In *2011 18th International Conference on High Performance Computing*. 1–11. https://doi.org/10.1109/HiPC.2011.6152725

[37] 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 (jun 2014), 25 pages. https://doi.org/10.1145/2611561

[38] Kalyan S Perumalla and Sudip K Seal. 2011. Discrete Event Modeling and Massively Parallel Execution of Epidemic Outbreak Phenomena. *SIMULATION: Transactions of The Society for Modeling and Simulation International* 88, 7 (1 2011). https://www.osti.gov/biblio/1037625

[39] D. M. Rao. [n. d.]. MUSE: A parallel Agent-based Simulation Environmentl. https://https://pc2lab.cec.miamioh.edu/muse/. [Online; accessed 11-Feb-2024].

[40] Dhananjai M. Rao. 2018. Performance comparison of Cross Memory Attach capable MPI vs. Multithreaded Optimistic Parallel Simulations. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Rome, Italy) *(SIGSIM-PADS '18)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/3200921.3200935

[41] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (mar 2011), 37–42. https://doi.org/10.1145/1964218.1964225

[42] Vinay Sachdev, M. Hybinette, and E. Kraemer. 2004. Controlling over-optimism in time-warp via CPU-based flow control. In *Proceedings of the 2004 Winter Simulation Conference, 2004.*, Vol. 1. 410. https://doi.org/10.1109/WSC.2004.

1371342
[43] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *International Symposium on Computer Architecture*. 475–486. https://doi.org/10.1145/2485922.2485963
[44] Richard T. Norris Sandra L. Glenn and Jude T. Sommerfeld. 1990. Discrete-event simulation in wastewater treatment. *Journal of Environmental Science and Health . Part A: Environmental Science and Engineering and Toxicology* 25, 4 (1990), 407–423. https://doi.org/10.1080/10934529009375567
[45] Nandakishore Santhi, Stephan Eidenbenz, and Jason Liu. 2015. The Simian concept: Parallel Discrete Event Simulation with interpreted languages and just-in-time compilation. In *2015 Winter Simulation Conference (WSC)*. 3013–3024. https://doi.org/10.1109/WSC.2015.7408405
[46] Markus Schordan, Tomas Oppelstrup, David Jefferson, Peter D. Barnes, and Dan Quinlan. 2016. Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Banff, Alberta, Canada) *(SIGSIM-PADS '16)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/2901378.2901394
[47] Lisa Sokol, Brian K. Stucky, and Vincent Shang-Shouq Hwang. 1989. MTW: A Control Mechanism for Parallel Discrete Simulation. In *Proceedings of the International Conference on Parallel Processing, ICPP '89, The Pennsylvania State University, University Park, PA, USA, August 1989. Volume 3: Algorithms and Applications*. Pennsylvania State University Press, 250–254.
[48] Tapas K. Som and Robert G. Sargent. 1998. A Probabilistic Event Scheduling Policy for Optimistic Parallel Discrete Event Simulation. *SIGSIM Simul. Dig.* 28, 1 (jul 1998), 56–63. https://doi.org/10.1145/278009.278016
[49] S. Srinivasan and P.F. Reynolds. 1995. NPSI adaptive synchronization algorithms for PDES. In *Winter Simulation Conference Proceedings, 1995*. 658–665. https://doi.org/10.1109/WSC.1995.478841
[50] Seng Chuan Tay, Yong Meng Teo, and Siew Theng Kong. 1997. Speculative Parallel Simulation with an Adaptive Throttle Scheme. *SIGSIM Simul. Dig.* 27, 1 (jun 1997), 116–123. https://doi.org/10.1145/268823.268909
[51] Roberto Toccaceli and Francesco Quaglia. 2008. DyMeLoR: Dynamic Memory Logger and Restorer Library for Optimistic Simulation Objects with Generic Memory Layout. In *2008 22nd Workshop on Principles of Advanced and Distributed Simulation*. 163–172. https://doi.org/10.1109/PADS.2008.23
[52] Tommaso Tocci, Alessandro Pellegrini, Francesco Quaglia, Josep Casanovas-Garcia, and Toyotaro Suzumura. 2017. ORCHESTRA: An asynchronous wait-free distributed GVT algorithm. In *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 1–8. https://doi.org/10.1109/DISTRA.2017.8167666
[53] SJ Turner and MQ Xu. 1992. Performance evaluation of the bounded Time Warp algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation*. 117–126.
[54] Jun Wang and Carl Tropper. 2007. Optimizing time warp simulation with reinforcement learning techniques. In *2007 Winter Simulation Conference*. 577–584. https://doi.org/10.1109/WSC.2007.4419650
[55] Jun Wang and Carl Tropper. 2009. Using genetic algorithms to limit the optimism in Time Warp. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*. 1180–1188. https://doi.org/10.1109/WSC.2009.5429634
[56] Barry Williams, Ali Eker, Kenneth Chiu, and Dmitry Ponomarev. 2021. High-Performance PDES on Manycore Clusters. In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Virtual Event, USA) *(SIGSIM-PADS '21)*. Association for Computing Machinery, New York, NY, USA, 153–164. https://doi.org/10.1145/3437959.3459252
[57] P.A. Wilsey. [n. d.]. Warped Simulation Kernel. https://github.com/wilseypa/warped2. [Online; accessed 11-Feb-2024].
[58] B. P. Zeigler, J. H. Hu, and J. W. Rosenblit. 1989. Hierarchical, modular modelling in DEVS-scheme. In *Proceedings of the 21st Conference on Winter Simulation* (Washington, D.C., USA) *(WSC '89)*. Association for Computing Machinery, New York, NY, USA, 84–89. https://doi.org/10.1145/76738.76749