Adaptive Methods for Irregular Parallel Discrete Event Simulation Workloads

Eric Mikida
University of Illinois at Urbana-Champaign
mikida2@illinois.edu

Laxmikant Kale
University of Illinois at Urbana-Champaign
kale@illinois.edu

ABSTRACT

Parallel Discrete Event Simulations (PDES) running at large scales involve the coordination of billions of very fine grain events distributed across a large number of processes. At such large scales optimistic synchronization protocols, such as TimeWarp, allow for a high degree of parallelism between processes, but with the additional complexity of managing event rollback and cancellation. This can become especially problematic in models that exhibit imbalance resulting in low event efficiency, which increases the total amount of work required to run a simulation to completion. Managing this complexity becomes key to achieving a high degree of performance across a wide range of models. In this paper, we address this issue by analyzing the relationship between synchronization cost and event efficiency. We first look at how these two characteristics are coupled via the computation of Global Virtual Time (GVT). We then introduce dynamic load balancing, and show how, when combined with low overhead GVT computation, we can achieve higher efficiency with less synchronization cost. In doing so, we achieve up to 2× better performance on a variety of benchmarks and models of practical importance.

ACM Reference format:
Eric Mikida and Laxmikant Kale. 2018. Adaptive Methods for Irregular Parallel Discrete Event Simulation Workloads. In *Proceedings of SIGSIM Principles of Advanced Discrete Simulation, Rome, Italy, May 23–25, 2018 (SIGSIM-PADS'18),* 12 pages.
https://doi.org/10.1145/3200921.3200936

1 INTRODUCTION

Discrete Event Simulation (DES) is a powerful tool for studying interactions in complex systems. These simulations differ from traditional time-stepped simulations in that the events being simulated occur at discrete points in time, which may not be uniformly distributed throughout the time-window being simulated. This makes DES ideal for modeling systems such as traffic flow, supercomputer networks, and integrated circuits [10, 16, 30]. Each of these modeling systems create events which a DES simulator executes in timestamp order. As these systems being simulated grow in size and complexity, the capability of sequential DES becomes insufficient, and the need for a more powerful simulation engine presents itself.

SIGSIM-PADS'18, May 23–25, 2018, Rome, Italy

ACM ISBN 978-1-4503-5092-1/18/05…$15.00
https://doi.org/10.1145/3200921.3200936

Parallel Discrete Event Simulations (PDES) can potentially increase the capability of sequential simulations, both by decreasing time to solution and by enabling larger and more detailed models to be simulated. However, this shift to a distributed execution environment raises the question of how to maintain timestamp order across multiple independent processes. Various synchronization protocols for PDES have been proposed and well studied to address this concern. In this paper, we focus on the Time Warp protocol originally proposed by Jefferson et al. [17]. In the Time Warp protocol, events are executed speculatively, and when a causality error between events is detected, previously executed events are rolled back until a point in time is reached where it is safe to resume forward execution.

The Time Warp protocol has been shown to achieve high performance and scalability at large scales in numerous instances [4, 21], however it is often the case that the models used in these studies naturally result in a uniform and balanced execution. In realistic models where the execution has a more irregular distribution, there is the possibility of cascading rollbacks as described in [12]. In this paper we aim to improve the performance and robustness of the Time Warp protocol by developing techniques that adaptively deal with irregularity and imbalance present in more realistic models operating at large scales. In order to do so we focus on two related characteristics that factor into a simulations performance: efficiency and synchronization cost, where efficiency is the ratio of committed events to total number of events executed.

The work and results presented in this paper utilize the Charm++ Adaptive Discrete Event Simulator (Charades). Charades was originally implemented as a Charm++ version of the Rensselaer Optimistic Simulation System (ROSS) [7]. ROSS has repeatedly demonstrated high performance at large scales, especially for network topology models [4, 23, 29]. In [21], Mikida et al. demonstrates the effectiveness of building ROSS on top of the Charm++ adaptive runtime system, where performance of the models tested was improved by $1.4 - 5\times$ due to a more effective management of fine-grained communication. Charades is based on this Charm++ version of ROSS, but with major changes to the underlying infrastructure to exploit more of the features of Charm++.

In Section 4, we study different configurations of the GVT computation and their effects on synchronization cost and event efficiency. We show a coupling between event efficiency and synchronization cost, which creates a tradeoff between the two quantities. In Section 5 we study load balancing as an additional way to control event efficiency, and show evidence that load balancing can be used to improve efficiency in GVT algorithms which sacrifice event efficiency for lower synchronization costs.

The main contributions we make in this paper are:

- Analysis of the tradeoff between synchronization cost and event efficiency present in blocking GVT algorithms
- Development of a scalable version of a non-blocking GVT algorithm for distributed systems
- Improvements to an existing distributed load balancing strategy for PDES use cases
- Analysis of load balancing as an additional efficiency control in concert with the non-blocking GVT algorithm

2 BACKGROUND AND RELATED WORK

In this section, we describe Charm++ and Charades, the software infrastructure used throughout the rest of this paper, as well as ROSS which the work was originally based upon. We also discuss related work in the area of GVT algorithms and load balancing in PDES.

2.1 CHARM++

Charm++ is a parallel programming framework built upon the notion of parallel asynchronous objects, called chares [1, 3]. Instead of decomposing applications in terms of cores or processes, Charm++ applications are decomposed into collections of chares. The Charm++ runtime system manages the placement of these chares and coordinates the communication between them.

Forward progress of an application is based on message-driven execution. Chares communicate via asynchronous one-sided messages, and only chares with incoming messages are scheduled for execution. This allows the runtime to adaptively schedule chares for execution based on the availability of work, and also leads to an adaptive overlap of communication and computation. This overlap relies on the fact that Charm++ applications are generally over-decomposed: there are many more chares in the application than there are cores.

Over-decomposition also gives the runtime system flexibility in location management of chares, and enables another important feature of Charm++: migration. Since the location and scheduling of chares is managed entirely by the runtime system, it has the freedom to migrate chares between different hardware resources as it sees fit. This enables features such as automated checkpoint/restart, fault-tolerance, and dynamic load balancing.

Charm++ has a robust dynamic load balancing framework, enabled by migratable chares, which allows applications to dynamically balance load across processes during execution. The load balancing framework monitors execution of chares as the application runs, and migrates chares based on the measurements it takes and the chosen load balancing strategy. There are many built-in load balancing strategies distributed with Charm++, as described in Section 5. These strategies vary widely in factors such as overhead incurred, whether or not communication is taken into account, and how the load information is aggregated across processes.

2.2 ROSS

ROSS is a massively parallel PDES implementation developed at RPI [6]. It utilizes the Time Warp protocol to synchronize optimistic simulations, where the specific mechanism for recovering from causality violations is based on reverse execution. Each LP has a forward event handler and a reverse event handler. When a causality violation occurs, the affected LPs execute reverse event handlers for events in the reverse order until they reach a safe point in virtual time to resume forward execution. ROSS is implemented on top of MPI, and demonstrates high performance and scalability on a number of models. Barnes et al. obtained 97× speedup on 120 racks (1.6 million cores) of Sequoia, a Blue Gene/Q system at Lawrence Livermore National Laboratory, when compared to a base run on 2 racks [4], and they have a number of publications showing highly scalable network models [23, 29].

In [21], ROSS is reimplemented on top of Charm++ in order to take advantage of the adaptive and asynchronous nature of the Charm++ runtime system. The primary difference between the MPI and Charm++ implementations is the encapsulation of LPs as chares in the Charm++ implementation, which enables the runtime system to adaptively schedule and migrate LPs during simulation execution. Suitability of the Charm++ programming model for PDES applications is evidenced by a decrease in the size of the code base by 50%, and by the increased performance and scalability for the PHOLD benchmark and the Dragonfly model simulating uniform traffic patterns. PHOLD event rates were increased by up to 40%, while the event rate for the Dragonfly model were reported to be up to 5× higher.

2.3 Charades

Charades is the simulation engine used for the experiments in this paper, and is an evolution of the Charm++ version of ROSS. The underlying infrastructure has been redesigned so that the GVT computation and the Scheduler are separated into two distinct and independent sets of chares. This allows for a modular class hierarchy for the GVT manager chares, which aids in the development of different GVT algorithms that can be selected from and instantiated at runtime. Furthermore, the separation of the Scheduler and GVT manager into independent sets of chares allows the runtime to more effectively overlap work between GVT computation and event execution. This is particularly important for the work in Section 4, where the work of the GVT computation is overlapped with event execution to decrease synchronization costs.

In addition to the redesigned infrastructure, Charades LPs are also migratable by virtue of being written as chares with routines for serialization and DE-serialization. This allows them to work within the Charm++ load balancing infrastructure. Furthermore, the LPs monitor various metrics that can be fed into the load balancing framework as a substitute for CPU time when determining the load of an LP.

2.4 GVT Computation

A significant amount of work has been devoted to the study of the GVT computation in optimistic simulations. The frequency at which it needs to occur, and the fact that it requires global information can cause it to become a major bottleneck if not synchronized properly. Non-blocking GVT algorithms have been shown to be successful on shared-memory systems by both Gomes et al. [15] and Fujimoto et al. [13]. On distributed systems, non-blocking algorithms which rely on atomic operations and machine clocks were studied by Chen et al. [9] and as part of the ROSS simulation system in [5].

Srinivasan's implementation in [27] relies on hardware support, specifically optimized for communicating global synchronization information. The SPEEDES simulation system also implemented an algorithm for computing the GVT without blocking event execution, however it did so by preventing the communication of new events until after the GVT computation [28].

For this work we have implemented a GVT algorithm targeted at high performance distributed computing environments based on the distributed snapshot algorithms proposed by Mattern and Perumalla [18, 25]. The GVT algorithm work is dynamically overlapped with other simulation tasks by the runtime system, including scheduling, event reception and execution, fossil collection, and rollbacks. We specifically look at the GVTs effects on event efficiency and synchronization cost. A more detailed description is given in Section 5 as well as results on 2048 processes.

2.5 Load Balancing

In [8], load balancing was shown to be effective for PDES applications running in a network of multi-user workstations. Similarly, [14] demonstrates dynamic load balancing on shared-memory processors by utilizing active process migration. Meraji et al. [20] shows benefits of load balancing for gate-level circuit simulations, and Deelman et al. utilizes load balancing for explicitly spatial models [11]. Here, we propose a more generalized load balancing framework, and one that is focused on HPC environments. We specifically look at load balancing as a tool to improve efficiency in conjunction with the GVT methods described in Section 4.

3 MODELS

To develop a better understanding of the impact of the techniques explored in this paper, we focus on variations of three diverse models with wide applicability: PHOLD, Dragonfly [22], and Traffic [2].

3.1 PHOLD

PHOLD is arguably the most commonly used micro-benchmark in the PDES community [4, 12, 31]. A basic PHOLD configuration is specified by 6 parameters: number of LPs (N), number of starting events (E), end time (T), lookahead (L), mean delay (M), and percentage of remote events (P). At the beginning of the simulation, N LPs are created, each with with an initial set of E events to be executed. During the execution of an event, a PHOLD LP creates and sends a new event to be executed in the future, and thus the simulation progresses. The execution of events is performed until the simulation reaches virtual time T.

For sending a new event, an event executed at time t creates an event that should be executed at time $t + d$, where d is the delay. Delay is calculated as the sum of the lookahead, L, and a number chosen randomly from an exponential distribution with mean M. With probability P, the destination of the new event is chosen randomly from a uniform distribution; otherwise, a self-send is done.

By default, PHOLD leads to a highly uniform simulation with each LP executing roughly the same number of events distributed evenly throughout virtual time. Hence, it is not a good representative of irregular and unbalanced models. We extend the base PHOLD with a number of parameters which control event and work distribution in order to make it a suitable representative of more complex simulation workloads. First, each LP is parameterized by the amount of time it takes to execute an event, which is controlled via a loop inside of forward execution that checks wall time until the set amount has elapsed. Secondly, the percentage of remote events, P, is now also set at the LP level rather than at the global level for the whole simulation, which means certain LPs can be set to do self-sends more frequently than others. By adjusting the distributions of these two parameters, an imbalance can be created both in the amount of work done by each LP as well as the number of events executed by each LP.

For the experiments in this paper we use four specific configurations for PHOLD. The baseline configuration (PHOLD Base) is a balanced configuration that has 64 LPs per process with $E = 16$, $L = 0.1$, $M = 0.9$ (for an expected delay of 1.0 per event), and $T = 1,024$. For the runs in this paper, this equates to $N = 131,072$ total LPs and $2,097,152$ initial events. For the base case, all LPs use $P = 50\%$ and each event is set to take approximately 1 nanosecond to process. The following unbalanced configurations modify the last two parameters for subsets of LPs to create different types of imbalance.

The work imbalance configuration (PHOLD Work) designates 10% of LPs as heavy LPs that take 10 nanoseconds to process each event instead of the baseline of 1 nanosecond. This results in a configuration with approximately twice the total work as PHOLD Base. The heavy LPs occur in a contiguous block starting at an arbitrarily chosen LP ID $2,165$. This offset was chosen so that the block of heavy LPs does not align evenly with the 64 LPs per process, creating more variation in processor loads that contain heavy LPs.

The event imbalance configuration (PHOLD Event) designates 10% of LPs as greedy LPs that have a remote percentage of 25% instead of 50%, meaning they are twice as likely to do a self-send than the remaining LPs. These greedy LPs are again set in a contiguous block starting at $2,165$.

The final configuration (PHOLD Combo) is a combination of the previous two configurations. 10% of the LPs starting at LP ID $2,165$ are both heavy and greedy, i.e. they perform ten times the work to process each event in comparison to other LPs, and are also twice as likely to do a self-send when compared to normal LPs.

3.2 Dragonfly

The Dragonfly model performs a packet-level simulation of the dragonfly network topology used for building supercomputer interconnects. We use a model similar to the one described in [21] and [22]. The Dragonfly model consists of three different types of LPs: routers, terminals, and MPI processes. The MPI process LPs send messages to one another based on the communication pattern being simulated. These messages are sent as packets through the network by the terminal and router LPs.

In this paper we experiment with four different communication patterns: Uniform Random (DFly UR) where each message sent goes to a randomly selected MPI process, Worst Case (DFly WC) where all traffic is directed to the neighboring group in the dragonfly topology, Transpose (DFly Trans) where message is sent to diagonally opposite MPI process, and Nearest Neighbor (DFly NN) where each message is sent to an MPI process on the same router.

The particular network configuration used in this paper has 24 routers per group and 12 terminals per model, which results in 6,936 routers connected to 83,232 terminals and MPI processes. Note that all these traffic patterns simulate balanced workloads with roughly the same number of MPI messages received by each simulated MPI process. Furthermore, only the network traffic is simulated, and compute time of each MPI process is ignored.

3.3 Traffic

The traffic model used in this paper is an extension of a traffic model available in the production version of ROSS [2], with added configuration parameters to create realistic scenarios. This model simulates a grid of intersections, where each intersection is represented as an LP. Each intersection is a four-way intersection with multiple lanes. Cars traveling through this grid of intersections are transported from one grid point to another through events. Different events in this simulation represent cars arriving at an intersection, departing from an intersection, or changing lanes.

Our baseline traffic configuration (Traffic Base) simulates $1,048,576$ cars in a 256×256 grid of intersections for a total of $65,536$ total LPs. Each car chooses a source and a destination uniformly at random from the set of intersections and travels from source to destination. From this base configuration we derive three unbalanced configurations by modifying the distribution of sources and destinations of the cars. This results in a different initial distribution of events per LP, as well as a distribution which changes over time as cars move throughout the grid.

The first configuration (Traffic Dest) represents what traffic may look like before a large sporting event, where a higher proportion of cars are traveling to a similar destination. In this particular configuration, 25% of the cars choose destinations in a 16×16 block area at the bottom right corner of the grid, and the rest of the cars choose their destination randomly.

The second configuration (Traffic Src) represents what traffic may look like after the sporting event, where a higher proportion of cars have a similar source. In this case, 10% of the cars start from a 16×16 block area at the top left corner of the grid and the rest of the cars are evenly distributed.

The final configuration (Traffic Route) is a combination of the previous two scenarios, where 10% of cars originate from the top left 16×16 area, and 25% of cars choose the bottom right 16×16 grid as their destination. This leads to similar patterns as the previous two distributions, but with the added effect of more cars following similar routes between their sources and destinations.

4 GLOBAL VIRTUAL TIME COMPUTATION

The calculation of the Global Virtual Time (GVT) is a critical part of an optimistic PDES simulation. The GVT is the latest virtual time that has been reached by every LP in the simulation, and therefore events prior to the GVT can be committed and their memory reclaimed via *fossil collection*. In order to prevent the simulator from running out of memory, the GVT must be computed frequently in order for fossil collection to keep up with the high rate of event creation. This, combined with the fact that the GVT computation requires global information, means that the GVT computation can quickly become a very costly bottleneck if not handled properly.

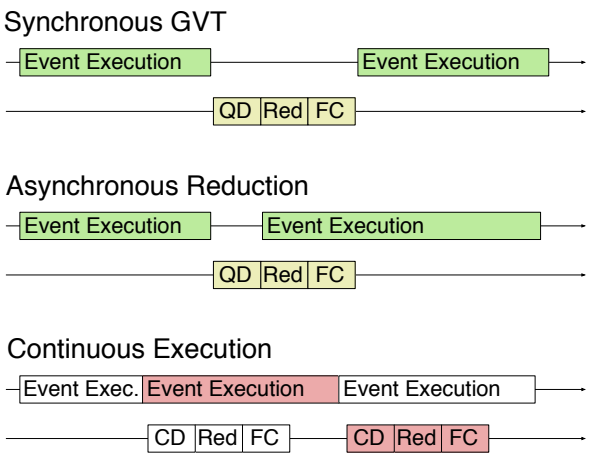

Figure 1: A depiction of three different algorithms for the GVT computation.

Furthermore, the synchronization required for the GVT computation can have additional side effects on event execution within the simulation.

In this section, we study two key components of a GVT calculation: trigger and algorithm. The trigger determines when and how often GVT is computed. The algorithm describes the actual process of computing the GVT, including how the computation is synchronized with the execution of LPs. The original implementation of Charades uses a count-based trigger and a fully synchronous GVT algorithm. This scheme is similar to the one implemented within ROSS and has achieved high performance and scalability as stated in Section 2. We will take this configuration as a baseline for comparing a virtual time-based trigger, and two GVT algorithms with varying degrees of asynchrony.

4.1 GVT Trigger

The trigger determines when to compute the GVT, and by extension, the state of the simulation when the GVT computation begins. The existing trigger is one based on event count. Specifically, each processor stops executing events and signals it is ready to take part in the GVT computation after executing N events, where N is a runtime parameter. This means that, in simulations where events take roughly the same amount of time to process, each processor will have roughly equal amounts of work during each GVT interval.

The second trigger we will be looking at is one based on a virtual time leash. With the leash-based trigger, each processor stops executing events and signals it is ready to take part in the GVT computation after progressing T units of virtual time from the previous GVT. The aim of this trigger is to keep each processor progressing at roughly the same rate, while placing less importance on keeping the amount of work done by each processor balanced.

4.2 GVT Algorithm

The GVT trigger controls when the scheduler signals it is ready to compute a GVT, however the behavior of the simulator during

the computation, including when event execution resumes, is controlled by the GVT algorithm itself. In the baseline synchronous algorithm event execution is blocked until the entire GVT computation is completed. It utilizes the Quiescence Detection library in Charm++ [26] to wait for all sent events to be received before computing the GVT. The algorithm consists of three parts, as shown in Figure 1 (top): Quiescence Detection (QD), an All-Reduce (Red), and Fossil Collection (FC). Since LPs are suspended for the entire GVT computation, this scheme theoretically has the highest overhead. However, since all LPs are suspended, this scheme also computes the maximal GVT since it knows the exact time reached by each LP. Computing the maximal GVT also results in freeing of maximal memory during fossil collection.

The following two algorithms preserve this basic structure, but attempt to exploit the asynchrony in Charm++ in various ways to decrease the time that LPs must block.

4.2.1 Asynchronous Reduction. The first technique we utilize to decrease the GVT synchronization cost exploits asynchronous reductions for the All-Reduce component of the computation. After quiescence is reached in the synchronous scheme, LPs are still blocked until after the reduction and the fossil collection occurs (Figure 1 (top)). This leaves many processors idle while the All-Reduce is being performed, even though the result of the All-Reduce is not required to execute more events. To address this shortcoming, we can restart event execution as soon as quiescence is reached and each processors local minimum is known, as shown in Figure 1 (mid). This allows the runtime system to adaptively overlap event execution with the All-Reduce and fossil collection, which reduces idle time by allowing processors to do meaningful work when they are not actively involved in the GVT computation.

4.2.2 Continuous Execution. The asynchronous reduction scheme described above still requires event execution to completely stop during quiescence detection, which, as we'll see in our experimental results, is the more costly part of the GVT synchronization. The second technique we explore aims at completely eliminating the need for LPs to block, allowing for continuous event execution throughout the simulation. In order to accomplish this, we use the concept of phase based completion detection to implement a scheme similar to the those described by Mattern and Perumalla [18, 24, 25].

In this algorithm, we use the completion detection library (CD) in Charm++ [1] as a replacement for the QD phase of the previous two algorithms. Unlike QD, completion detection monitors only a subset of messages, and is triggered once all of those messages have been received. By carefully tagging subsets of events and running multiple CD instances independently, we enable continuous execution of events overlapped with the GVT computation.

The execution of the continuous scheme is split into two recurring phases, which we refer to as the red phase and the white phase, as shown in Figure 1 (bot). Each phase is managed by a single instance of the CD library. When an event is sent in the white phase, it is tagged as a white event and the white CD instance is informed so it can increment its count of sent events. Similarly, events sent during the red phase are tagged as red. When an event is received, the corresponding CD instance is informed based on the event's tag so it can increment its count of received events. When it is time to switch from one phase to the next, the CD instance for the current phase is informed that no more events will be sent, which triggers a series of reductions for that CD instance to determine the total number of events sent and received. Once the CD instance detects that the number of sent events equals the number of received events, the GVT can be computed with an All-Reduce of the local minima followed by fossil collection. Throughout the entire process, events are still being executed in the next phase, and the work from event execution is overlapped with work for the CD library, GVT computation, and fossil collection.

One additional difference in this new scheme is how the GVT is computed from local minima. In the previous two schemes, since total quiescence is reached, all events have been received and the GVT is simply the minimum of all received events. However, the continuous scheme never requires full quiescence so there may be events in flight that could affect the GVT. In particular, when switching from one phase to another, for example from white to red, the simulator is waiting on all white events to arrive but also sending outgoing red events. Once all white events have been received the simulator can guarantee the minimum timestamp of red events sent in the future, however it must also take into account red events that have already been sent. Since it has no guarantee that any of the previously sent red events have been received it must track these events from the sender side. Specifically, between the time that the phases are switched and events for the previous phase have all been received, each processor must track the minimum timestamp of outgoing events and take that into account when contributing to the minimum All-Reduce.

	Synchronous		Continuous	
Model	**Count**	**Leash**	**Count**	**Leash**
PHOLD Base	97%	98%	95%	95%
PHOLD Work	75%	76%	52%	52%
PHOLD Event	54%	84%	60%	60%
PHOLD Combo	54%	93%	31%	31%
DFly Uniform	74%	62%	36%	36%
DFly Worst	39%	91%	2%	2%
DFly Trans	31%	85%	27%	27%
DFly NN	68%	93%	67%	67%
Traffic Base	46%	96%	54%	55%
Traffic Src	12%	97%	16%	16%
Traffic Dest	51%	96%	52%	52%
Traffic Route	11%	97%	15%	15%

Table 1: Table comparing event efficiencies between the two different triggers. The leash trigger results in higher efficiency for almost all model configurations.

4.3 Experimental Results

The following experiments were done on all models and configurations described in Section 3, using each combination of GVT trigger and algorithm. Runs were done on 64 nodes (2,048 cores) of the Blue Waters system at NCSA. The parameters for each trigger were tuned to achieve the best performance. Multiple runs of each

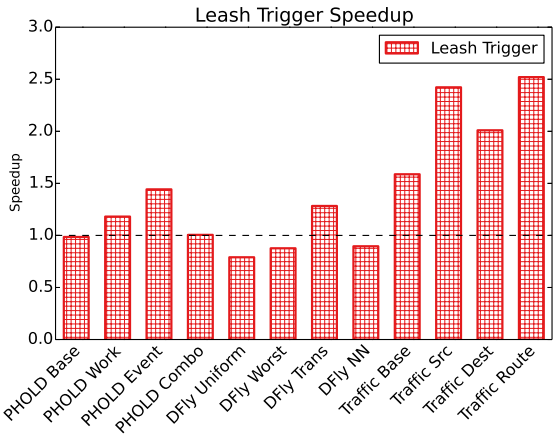

Figure 2: Speedup of the leash-based trigger over the count-based trigger, both using the synchronous GVT algorithm.

Model	Sync	Async Reduction
PHOLD Base	2.75s	2.32s (16%)
PHOLD Work	9.23s	8.73s (5%)
PHOLD Event	5.96s	5.50s (8%)
PHOLD Combo	25.14s	24.56s (2%)
DFly Uniform	3.42s	3.79s (-11%)
DFly Worst	4.73s	3.07s (35%)
DFly Trans	7.51s	5.51s (27%)
DFly NN	0.59s	0.28s (53%)
Traffic Base	5.27s	4.16s (21%)
Traffic Src	22.20s	20.80s (6%)
Traffic Dest	8.30s	7.15s (14%)
Traffic Route	26.35s	24.66s (6%)

Table 2: Table comparing the amount of time LPs spend blocking (in seconds) under the synchronous and asynchronous GVT reduction algorithms, as well as the percent improvement when using the asynchronous reduction.

configuration were performed, but in all cases variance between runs was negligible.

4.3.1 Leash-Based Trigger. In analyzing the effects of the different GVT configurations, we look both at synchronization cost and event efficiency and how each one impacts event rate. First, analyzing each GVT trigger under the baseline synchronous scheme demonstrates the impact that the GVT trigger has on overall performance. After tuning the count-based trigger for the highest event rate, the virtual time leash for the leash-based trigger was initially chosen so that the total number of GVT computations for the simulation would be similar to what was required by the count metric. Then the leash was increased or decreased accordingly to find optimum performance. In almost all cases, the best performance came by decreasing the leash, which resulted in more GVT computations and a higher synchronization cost, but a higher event efficiency.

Table 1 shows the event efficiency for the two triggers, with the leash-based trigger achieving higher efficiency for all but one model configuration. In many cases the improvement in efficiency is significant, especially in the Traffic model where the leash-based trigger achieves at least 96% efficiency in all configurations. The models that see less benefit from the leash-based trigger are those where the distribution of events across LPs is roughly uniform: PHOLD Base, PHOLD Work, and DFly Uniform. The increased efficiency means fewer rollbacks, and so each model has to execute fewer events when running to completion under the leash-based trigger. Figure 2 shows the speedup of the leash-based trigger when compared to the count-based trigger, where we see significant speedups for most of the model configurations. In particular, the Traffic model sees over 2x speedup for its unbalanced configurations due to the drastic improvements to efficiency. For model configurations where the efficiency between the two triggers was comparable, the event rates are similar for both and may even slightly favor the count-based trigger. In these cases, the leash-based trigger often spent more time where LPs were blocked waiting for the GVT, either due to the leash trigger computing GVT more often, or due to an imbalance in the amount of time spent blocking on the GVT since the number of events computed on each PE varies per GVT in the leash-based trigger.

4.3.2 Asynchronous Algorithms. Where the trigger had a more profound effect on event efficiency, the primary impact of the different GVT algorithms is in reducing the synchronization cost. Due to the fact that the leash-based trigger provided comparable or better performance in most cases, we will first look at the effects of each GVT algorithm when using the leash trigger. Table 2 shows the amount of time LPs spent blocking on the GVT computation for the synchronous algorithm and the asynchronous reduction algorithm. Time spent blocking when using the continuous algorithm is zero in all cases, so is not shown in the table. The table also shows the percent change, and we see a reduction in time blocking by over 20% in most of the Dragonfly models, and over 10% in PHOLD Base, Traffic Base, and Traffic Dest. However this also highlights the fact that the majority of the time spent blocking on the GVT comes from Quiescence Detection. In Figure 3 (right), which plots the speedup of the two GVT algorithms when compared to the synchronous algorithm, we see that in most cases the speedup from the asynchronous reduction scheme is comparable to the reduction in blocking time. The results are especially pronounced in the Dragonfly and Traffic models, which have a lower ratio of event work to GVT work, whereas the PHOLD models generate more events and a smaller fraction of them are able to be executed during the time freed up by the asynchronous reduction.

Figure 3 shows the effects of the continuous algorithm which eliminates all LP blocking during the GVT. In cases where the asynchronous reduction was able to improve event rate by reducing synchronization cost, we see that the continuous scheme was able to be even more effective, achieving up to 1.5× speedup for DFly Uniform and Traffic Base, and 3× speedup for DFly NN. For DFly Worst, PHOLD Event, and PHOLD Combo, the efficiency is significantly worse, which outweighs the benefits of the reduced synchronization cost.

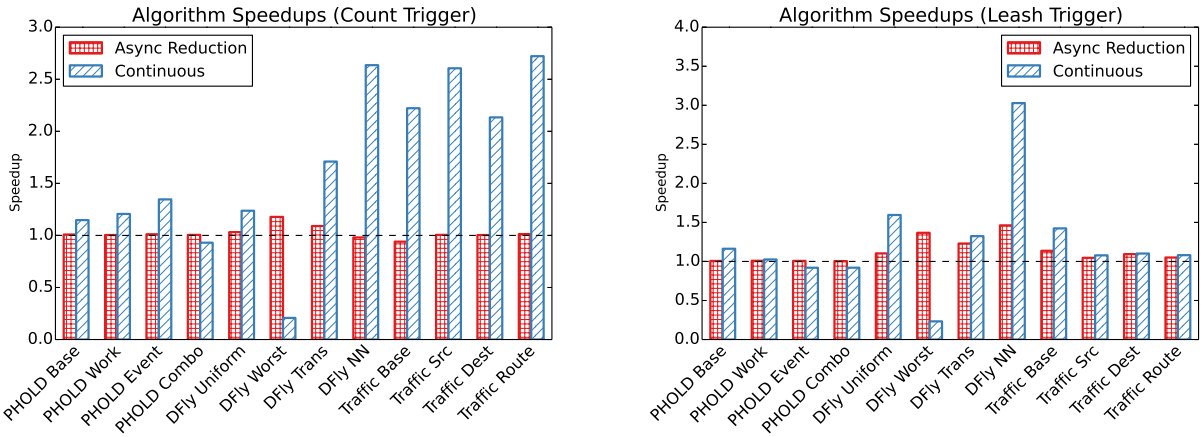

Figure 3: The relative speedup of each GVT algorithm when compared to the synchronous version.

When looking at how the different algorithms affect the count-based trigger in Figure 3 (left), we see largely the same patterns but with much larger speedups for the continuous algorithm. Figure 4 plots the actual event rates of the continuous algorithm for both triggers, and shows the the performance is nearly identical regardless of the trigger used. This is further backed up by nearly identical efficiencies under each trigger as shown in Table 1. This reveals an important side-effect of the other two GVT algorithms, which is that by blocking the LPs during at least part of the GVT computation, the amount of optimistic execution allowed by the simulator is bounded by the GVT computation. LPs can not get too far ahead due to the fact that either after N events, or T units of virtual time depending on the trigger, they must block while other LPs are allowed to catch up. The effects of this already showed up when comparing the two triggers under the synchronous algorithm. LPs in the count trigger can still get arbitrarily far ahead in virtual time depending on the distribution of events, but under the leash trigger LPs cannot, which keeps LPs closer together in virtual time reducing the likelihood of causality violations. Once that artificial bounding is removed from the GVT computation, the rate at which LPs progress through virtual time is completely unbounded and solely up to the characteristics of the models themselves. This results in lower efficiencies when compared to the synchronous scheme, which limits the benefits from the reduced synchronization costs. This limit is less detrimental under the count-based trigger due to the fact that it already has relatively lower efficiencies when compared to the leash-based trigger.

4.4 Summary

Analysis of the above GVT techniques shows a clear coupling between event efficiency and synchronization costs. The algorithms that block event execution while waiting for quiescence achieve a higher event efficiency by bounding the amount of execution. Tighter leashes further improve efficiency while simultaneously causing more frequent GVT computations, creating a tradeoff between efficiency and synchronization. The continuous algorithm

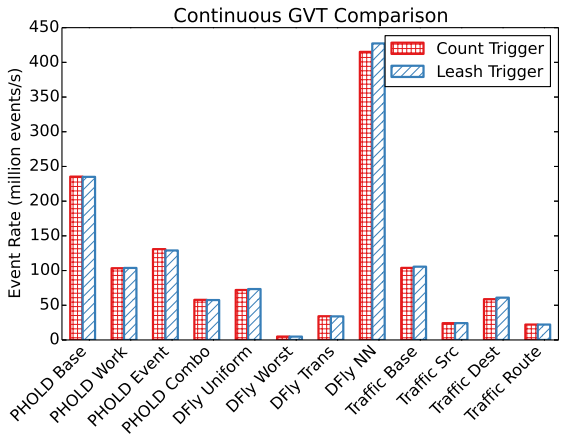

Figure 4: Comparison of the event rates for the continuous GVT algorithm under both triggers.

attempts to completely remove the synchronization cost, which in turn results in lower efficiency from the now unbounded optimism. In the next section we look at a method for improving efficiency independent of the GVT computation in order to decouple efficiency and synchronization.

5 LOAD BALANCING

In the previous section, we explored a tradeoff between synchronization cost and event efficiency. In many cases, enforcing more synchronization was able to increase the event efficiency, and resulted in a higher event rate. However, after a certain point, the synchronization cost outweighs the benefits gained from higher efficiency. Furthermore, as shown with the continuous GVT scheme, completely removing the synchronization cost of the global barrier may result in even higher performance despite large decreases in efficiency. In this section, we explore another method for affecting

the efficiency of simulations: dynamic load balancing. When the distribution of events across processors is unbalanced, processors with little work can run far ahead of the rest, and be forced to rollback frequently as a result. Improving efficiency with load balancing will allow us to lower the synchronization cost of the GVT computation, while still maintaining a certain degree of efficiency.

LPs in Charades are implemented as chares in Charm++. This allows the runtime system to manage the location of LPs as well as migrate them for purposes such as dynamic load balancing. During simulation execution, the load of each LP can be measured as CPU time automatically by the runtime system, or by metrics specific to Charades. Once the simulator is ready to perform load balancing, the load statistics are collected and LPs redistributed according to the particular load balancing strategy being used. For the experiments in this section we will be working with two different strategies: GreedyLB and DistributedLB.

GreedyLB is a centralized strategy that collects profiling information about all LPs on a single processor before redistributing them using a greedy algorithm. It iteratively assigns the most heavy unassigned LP to the least loaded processor until LPs have been assigned. As this can result in almost every LP being migrated, we also test a variation of GreedyLB that attempts to limit the number of migrations when reassigning the LPs.

DistributedLB is a fully distributed strategy that utilizes probabilistic migration and a gossip protocol as described in [19] to minimize the overhead of load balancing. First, a global reduction is done to determine the average load of all processors. Information about which processors are under-loaded is then propagated throughout the system via "gossip" messages. Once gossip is complete, the overloaded processors asynchronously attempt to shift some of their load to the under-loaded ones in a probabilistic fashion. In initial experiments, this did little to affect the performance of our simulations due to very few objects successfully migrating. For this work, we have modified the DistributedLB algorithm by breaking up the load transferring step into multiple phases. In earlier phases, only the most overloaded processors have the opportunity to shift their load, which makes it less likely that their attempts will fail due to other processors transferring load first. In subsequent phases, the threshold for which processors can shift load is relaxed. This prioritizes reducing the ratio of max to average load more aggressively than the original implementation.

In addition to the two different strategies, we examine two different load metrics for determining the load of an LP. The first is CPU time, which is automatically measured by the runtime system. The second is the number of committed events. Using committed events as a measure of load aims to ignore incorrect speculation when balancing load by only focusing on meaningful work done by each LP.

5.1 Experimental Results

The following experiments were run using the six unbalanced configurations of PHOLD and Traffic, on 64 nodes (2,048 cores) of Blue Waters. Due to the fact that the communication patterns in Dragonfly were uniformly balanced in the amount of work for each MPI process, load balancing had little effect on performance and is therefore omitted from this section. In each case, load balancing was triggered after the completion of a GVT computation and fossil collection had occurred. This allowed for the most memory within the LPs to be freed before migrating them to minimize the migration footprint. The particular GVT which triggered load balancing was determined via some manual tuning, and was always somewhere within the first 10% of a simulation in order to allow enough time for load statistics to be gathered. In all the results presented, statistics shown are for the entire simulation run, which includes execution both before and during the load balancing phase. Initially we focus our analysis on the isolated effects of load balancing under the synchronous GVT algorithm. Furthermore, we focus on the best load balancing strategy for each model only, allowing more space to analyze the effects of load balancing and the different load balancing metrics. For PHOLD, DistributedLB performed the best, where as GreedyLB was the best load balancer for the traffic model. As stated in the previous section, variation between multiple runs was negligible.

Figure 5 shows speedup of each model configuration with each load measure, compared to results without load balancing. With the exception of Traffic Dest with the count-based trigger, there is an increase in performance in each model configuration with at least one of the load metrics. The two primary causes for the increased event rate are improved balance of work across processors, and increased event efficiency. The magnitude of each effect depends heavily on the trigger used, where load balancing serves to improve on the shortcomings of that particular metric.

5.1.1 Synchronization Cost and Balance. In the case of Charades, the work we are interested in balancing is the execution of events in between GVT computations. Between two GVT computations, each processor has a set amount of work which, based on the trigger, is either a specified number of events to execute, or an amount of virtual time to traverse. Processors that finish their work early due to an imbalance in the number of events, weight of events, or a higher efficiency resulting in less rollbacks and re-execution will therefore reach the next GVT computation quicker and begin blocking before other processors. Figure 6 plots the min, max, and mean time spent blocking on the GVT across all processors. In the cases with no load balancing, many models exhibit a wide range of times indicative of imbalance across processors.

For the leash-based trigger, load balancing is able to decrease the range of times, as well as lowering the max and average, meaning less overall time spent waiting on the GVT. For configurations with work imbalance, namely PHOLD Work and PHOLD Combo, we see that the CPU load metric is particularly effective at decreasing wait times. For other models where the imbalance is primarily due to different distributions of events, both metrics are effective at both lowering the amount of time waiting as well as shrinking the range of times. These effects are prominent with the leash-based trigger due to the fact that the amount of work per GVT interval that is enforced by the trigger is not necessarily equal across all processes and depends on the distribution of events. Load balancing attempts to balance the amount of work required for each processor to traverse each interval, where the leash-based trigger still plays the role of keeping each processors rate of progress through virtual time balanced. This results in less variation in how long it takes each processor to get from one GVT computation to the next.

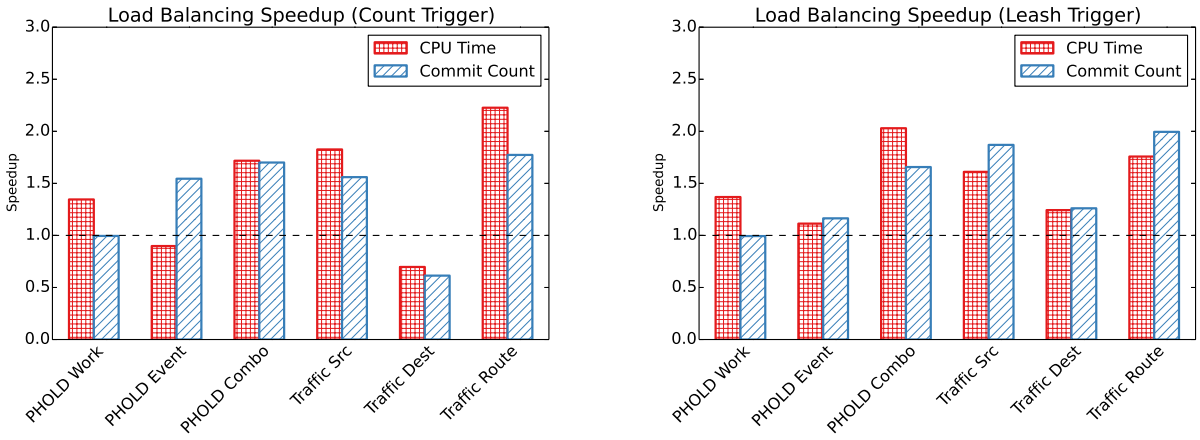

Figure 5: Speedup of each model configuration with each load balancing metric, compared to configurations with no load balancing.

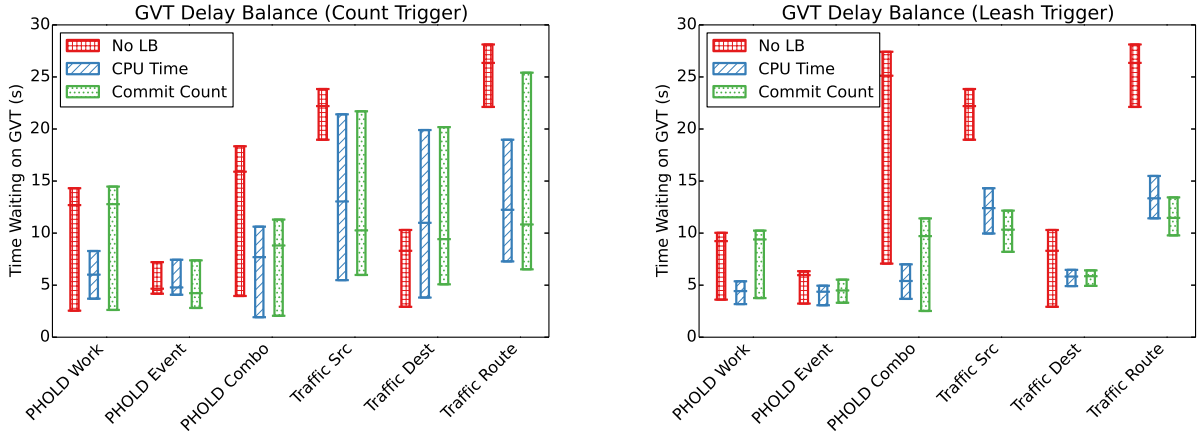

Figure 6: Min, max, and mean time across all processors spent blocking on the GVT computation. Processors with very little work in a given interval will block longer waiting for others to catch up.

The left-hand side of figure 6 shows the GVT wait time for the count-based metric. Load balancing is not nearly as effective at shrinking the range of times, and in the traffic models we actually see a wider range. The count-based metric inherently enforces balanced GVT intervals for models like Traffic where all events take roughly the same amount of time. However, load balancing still manages to decrease the overall amount of time waiting on the GVT in almost every case. This is due to a net gain in efficiency when using load balancing as shown on the left-hand side of figure 7. The improved efficiency results in less overall work to complete the simulation and therefore fewer GVT calculations.

5.1.2 Efficiency. When comparing the efficiency for the two different triggers in figure 7, we see that the effectiveness of load balancing is the opposite of what we saw when looking at the GVT wait time. In this case, since the count-based trigger does little to enforce high efficiencies, load balancing is far more successful at increasing efficiency for these configurations. Every model but Traffic Dest saw an increase in efficiency, with the results being particularly good for PHOLD Event with the commit metric, and PHOLD Combo with both. The effects are less pronounced for the Traffic models where we only see modest gains in efficiency. While PHOLD has no communication locality, as every LP is sending to every other LP with the same probability, migrating LPs did not effect communication costs. For Traffic however, there is high degree of communication locality, so while migrating objects may provide a better balance of the number of events executed and committed across processors, it also causes higher communication costs between heavily communicating LPs. For the leash-based trigger, which enforced very high efficiency in the Traffic model, we see

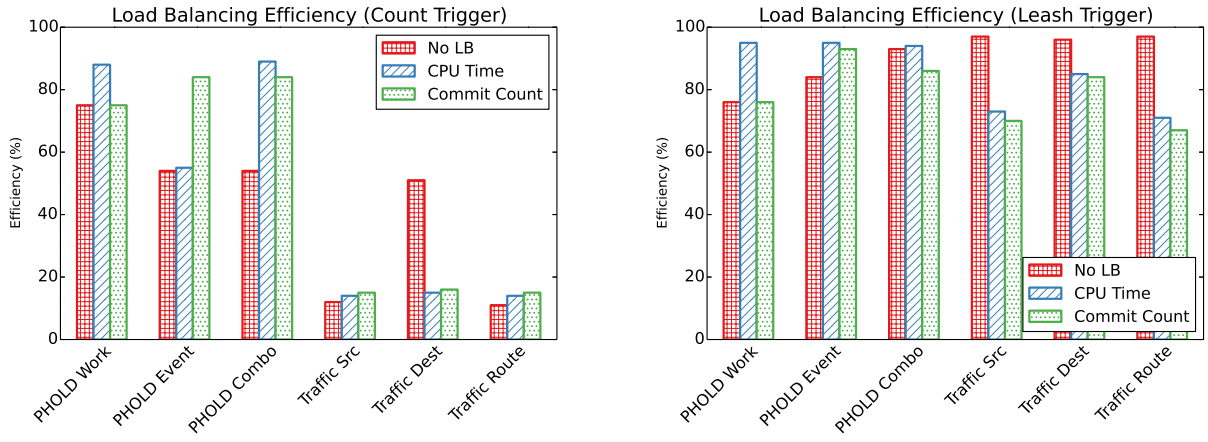

Figure 7: Efficiency of each model with and without load balancing.

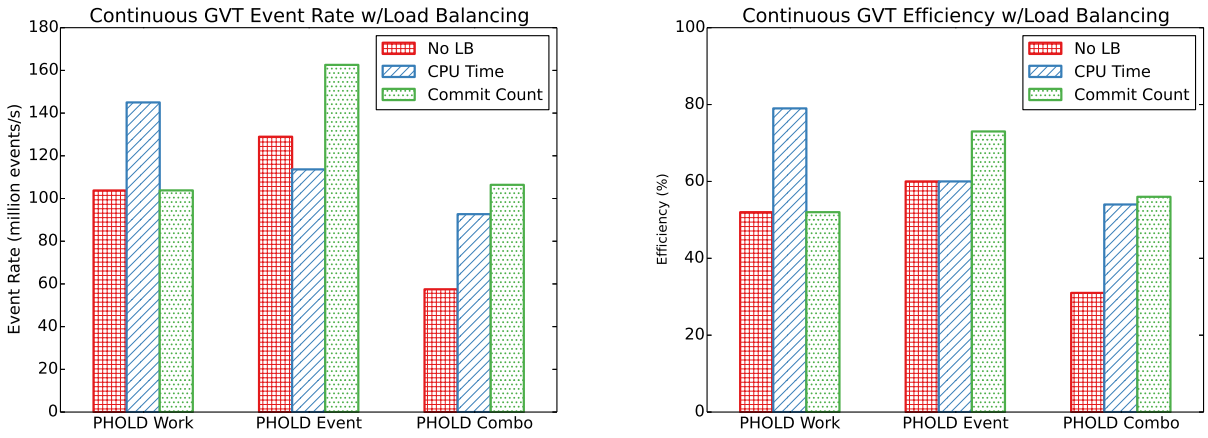

Figure 8: Comparison of the continuous GVT algorithm with and without load balancing.

that load balancing actually lowers the efficiency due to more rollbacks and event cancellations between the heavily communicating LPs.

5.1.3 Combining Load Balancing and Continuous GVT. The fact that load balancing is able to improve efficiency is especially important when considering the fact that the limiting factor in performance of the continuous GVT algorithm from Section 4 was a decrease in event efficiency. Figure 8 shows the event rates for the continuous GVT algorithm when load balancing is introduced. We see event rate improved in all cases by up to 2×. All three have at least 1.6× speedup over the original baseline configurations, and PHOLD Work and PHOLD Event achieve their highest event rates for any configuration studied in this paper. The efficiency plot in figure 8 further reinforces the impact of the different load metrics. When all other synchronization from the GVT is removed, it becomes clear that when addressing different types of imbalance the correct load metric must be chosen in order to achieve the best efficiency. For PHOLD Work, where the primary cause of imbalance is the amount of work per event, the CPU load metric improved efficiency by a factor of around 1.6× but the commit count metric had no effect at all. The opposite is true for PHOLD Event where the primary cause of imbalance is an uneven distribution of events. For PHOLD Combo, both metrics do well but it is conceivable that a combination of the two metrics may be even more effective.

Unfortunately, in the case of the Traffic model, the efficiency becomes much too low in the continuous case. Combining that with the communication locality issue described earlier means that the continuous scheme with load balancing is ineffective, and the rollbacks and especially cancellation events cascade out of control. This points to a need for a communication aware load balancing strategy, which would keep tightly coupled LPs in close proximity in an attempt to minimize the number of rollbacks and event cancellations between them.

5.2 Summary

By introducing automated load balancing into Charades, we are able to increase the event rate of unbalanced models by improving both balance and efficiency. Depending on the type of imbalance in the model, different metrics for defining the load of an LP will improve the capability of the load balancing framework, as each metric addresses different aspect of a simulation. Furthermore, the effect of load balancing on event efficiency comes without additional synchronization cost, so it becomes particularly effective when combined with the continuous GVT from Section 4. However, load balancing was not effective at addressing efficiency in models with balanced workloads such as Dragonfly. Furthermore, for the Traffic model, load balancing was less effective due to high communication locality and much lower efficiencies. We leave it as future work to further explore how careful use of different load metrics and balancing strategies can detect and deal with event efficiency independent from load balance. We also plan to develop light-weight distributed strategies for better handling models with high communication locality in order to handle negative side effects such as an increase in cancellation events.

6 CONCLUSION

Software that enables scalable execution of discrete event simulation is desirable in several domains. However the irregular, dynamic and fine-grained nature of many PDES models makes it difficult to scale. In particular, the event efficiency in such models can be difficult to manage and often requires the need for high synchronization costs. In this paper, we evaluated several techniques for managing the tradeoff between synchronization cost and event efficiency. We also provide a scalable implementation of a non-blocking GVT algorithm to remove the synchronization cost entirely. Up to 2× better performance is shown for a variety of models running on 2048 processors.

Furthermore, we explore dynamic load balancing as a technique to manage event efficiency, without requiring explicit synchronization from the GVT computation. In unbalanced model configurations, load balancing is shown to increase both event rate and event efficiency. By combining load balancing with the continuous GVT algorithm, even higher performance is achieved for our unbalanced version of the PHOLD benchmark than with any of the other techniques in this paper, and shows that load balancing can mitigate efficiency loss independently of the GVT algorithm used.

REFERENCES

[1] The charm++ parallel programming system manual. http://charm.cs.illinois.edu/manuals/html/charm++/manual.html.

[2] Ross source code on github. https://github.com/carothersc/ROSS, visited 2016-03-20.

[3] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. SC, 2014.

[4] P. D. Barnes, Jr., C. D. Carothers, and D. R. e. a. Jefferson. Warp speed: Executing time warp on 1,966,080 cores. In *Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS, pages 327–336, New York, NY, USA, 2013.

[5] D. Bauer, G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman. Seven-o'clock: A new distributed gvt algorithm using network atomic operations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, PADS '05, pages 39–48, Washington, DC, USA, 2005. IEEE Computer Society.

[6] D. W. Bauer Jr., C. D. Carothers, and A. Holder. Scalable time warp on blue gene supercomputers. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pages 35–44, Washington, DC, USA, 2009. IEEE Computer Society.

[7] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: A high-performance, low-memory, modular Time Warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[8] C. D. Carothers and R. M. Fujimoto. Efficient execution of time warp programs on heterogeneous, now platforms. *IEEE Trans. Parallel Distrib. Syst.*, 11(3):299–317, Mar. 2000.

[9] G. G. Chen, Boleslaw, and K. Szymanski. Time quantum gvt: A scalable computation of the global virtual time in parallel discrete event simulations.

[10] N. Choudhury, Y. Mehta, T. L. Wilmarth, E. J. Bohm, and L. V. . Kalé. Scaling an optimistic parallel simulation of large-scale interconnection networks. In *Proceedings of the Winter Simulation Conference*, 2005.

[11] E. Deelman and B. K. Szymanski. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 46–53, May 1998.

[12] R. M. Fujimoto. Performance of time warp under synthetic workloads. Distributed Simulation Conference, 1990.

[13] R. M. Fujimoto and M. Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Trans. Model. Comput. Simul.*, 7(4):425–446, Oct. 1997.

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

[15] Z. X. F. Gomes, B. Unger, and J. Cleary. A fast asynchronous gvt algorithm for shared memory multiprocessor architectures. *SIGSIM Simul. Dig.*, 25(1):203–208, July 1995.

[16] E. J. Gonsiorowski, J. M. LaPre, and C. D. Carothers. Improving accuracy and performance through automatic model generation for gate-level circuit pdes with reverse computation. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 87–96, New York, NY, USA, 2015. ACM.

[17] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. In *Proceedings of the Conference on Distributed Simulation*, pages 63–69, July 1985.

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

[19] H. Menon and L. Kalé. A distributed dynamic load balancer for iterative applications. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 15:1–15:11, New York, NY, USA, 2013. ACM.

[20] S. Meraji, W. Zhang, and C. Tropper. On the scalability and dynamic load-balancing of optimistic gate level simulation. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29(9):1368–1380, Sept. 2010.

[21] E. Mikida, N. Jain, E. Gonsiorowski, P. D. Barnes, Jr., D. Jefferson, C. Carothers, and L. V. Kale. Towards pdes in a message-driven paradigm: A preliminary case study using charm++. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '16. ACM, May 2016.

[22] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 366–376, Nov 2012.

[23] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. Using massively parallel simulation for mpi collective communication modeling in extreme-scale networks. In *Proceedings of the 2014 Winter Simulation Conference*, WSC '14, pages 3107–3118, Piscataway, NJ, USA, 2014. IEEE Press.

[24] K. S. Perumalla, A. J. Park, and V. Tipparaju. Gvt algorithms and discrete event dynamics on 129k+ processor cores. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–11, Dec 2011.

[25] K. S. Perumalla, A. J. Park, and V. Tipparaju. Discrete event execution with one-sided and two-sided gvt algorithms on 216,000 processor cores. *ACM Trans. Model. Comput. Simul.*, 24(3):16:1–16:25, June 2014.

[26] A. B. Sinha, L. V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 1993.

[27] S. Srinivasan and P. F. Reynolds, Jr. Non-interfering gvt computation via asynchronous global reductions. In *Proceedings of the 25th Conference on Winter Simulation*, WSC '93, pages 740–749, New York, NY, USA, 1993. ACM.

[28] J. S. Steinman, C. A. Lee, L. F. Wilson, and D. M. Nicol. Global virtual time and distributed synchronization. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*, PADS '95, pages 139–148, Washington, DC, USA, 1995. IEEE Computer Society.

[29] N. Wolfe, C. D. Carothers, M. Mubarak, R. Ross, and P. Carns. Modeling a million-node slim fly network using parallel discrete-event simulation. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '16, pages 189–199, New York, NY, USA, 2016. ACM.

[30] Y. Xu, W. Cai, H. Aydt, M. Lees, and D. Zehe. An asynchronous synchronization strategy for parallel large-scale agent-based traffic simulations. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 259–269, New York, NY, USA, 2015. ACM.

[31] S. B. Yoginath and K. S. Perumalla. Optimized hypervisor scheduler for parallel discrete event simulations on virtual machine platforms. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SimuTools '13, pages 1–9, ICST, Brussels, Belgium, Belgium, 2013. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).