An Adaptive Non-Blocking GVT Algorithm

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

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

ABSTRACT

In optimistic Parallel Discrete Event Simulations (PDES), the Global Virtual Time (GVT) computation is an important aspect of performance. It must be performed frequently enough to ensure simulation progress and free memory, while still incurring minimal overhead. Many algorithms have been studied for computing the GVT efficiently under a variety of simulation conditions for a variety of models. In this paper we propose a new GVT algorithm which aims to do two things. First, it incurs a very low overhead on the simulation by not requiring the simulation to block execution. Secondly, and most importantly, it has the ability to adapt to simulation conditions while it's running. This allows it to perform well for a variety of models, and helps remove some burden from developers by not requiring intensive tuning.

ACM Reference Format:
Eric Mikida and Laxmikant Kale. 2019. An Adaptive Non-Blocking GVT Algorithm. In *SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS '19), June 3–5, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3316480.3322896

1 INTRODUCTION

Parallel Discrete Event Simulation (PDES) is a complex topic, with many proposed solutions for various problems. In this paper, we will be focusing on distributed optimistic simulations which utilize Jefferson's TimeWarp protocol to synchronize the simulation. In particular, we look at improving the computation of the Global Virtual Time (GVT). In optimistic simulations, the GVT at a particular real time, t, is defined as the minimum of every processes local times and the times of all in-flight events [9]. Computing the GVT is critical for optimistic simulations because it is the only way for a simulator to determine simulation progress, allows for non-reversible event side effects to be committed, and allows the simulator to reclaim event memory from committed events. Because of its importance to the simulation, the GVT must be computed frequently, but without incurring too much overhead.

The GVT is a global property of the entire simulation state. When running on a distributed memory machine, this state is spread over multiple processes (running on different nodes). If the engine were *omniscient*, it would be able to determine the exact GVT at any given instant. However, without an omniscient simulation engine, we must rely on different techniques to compute either the exact GVT, or an approximation of it. A practical implementation can either stop the simulation entirely to calculate an exact GVT, or approximate a conservative lower bound on the GVT during simulation execution. Such a lower bound on GVT will still allow for the simulation to commit events and reclaim memory, albeit less effectively than if the actual GVT was used.

In this paper, we propose a new distributed algorithm to estimate the GVT which we call the Adaptive Bucketed GVT algorithm. It operates alongside event communication and computation by dividing execution of the simulation into buckets of virtual time. Furthermore, it adapts to simulation conditions in determining the frequency and scope of the computation. The algorithm is described in detail in Section 3.

In order to show effectiveness of our algorithm we directly compare it against a standard blocking algorithm, similar to the one used in the highly scalable Rensselaer Optimistic Simulation System (ROSS) [2, 3, 7], and the implementation of Mattern's non-blocking GVT algorithm [9]. All algorithms evaluated in this paper are implemented in the Charades PDES simulator built on top of the Charm++ runtime system [11]. We compare performance of each algorithm on a variety of model configurations in Section 4.

Finally, when evaluating non-blocking GVT algorithms, a common pattern that appears is a low event efficiency. Here, event efficiency is defined as the number of events committed divided by the total number of events executed. Improving event efficiency results in less work and communication done by the simulator, and may therefore be an avenue to improving overall simulation performance depending on what other simulation characteristics are perturbed. Because of this, and observations made about techniques used in the SPEEDES simulation engine [17, 18], in Section 5 we look at one potential way to increase event efficiency using our GVT algorithm. With its already introspective nature, we look to use the data collected by the GVT algorithm to adaptively throttle event communication for events which may be likely to be rolled back. The resulting decrease in events and anti events sent is able to improve performance in certain cases, and opens up other potential areas of research.

2 BACKGROUND AND RELATED WORK

This section describes previous research in the area of GVT algorithms, and briefly discusses the simulator and models used in this paper to evaluate each algorithm.

2.1 Charades

Charades is a PDES simulation engine originally based on principles in ROSS, and implemented on top of the Charm++ runtime system [10, 11]. The driving principle behind the simulator is its asynchronous message-driven design. All messaging between Logical Processes (LPs) is handled by the Charm++ runtime system,

SIGSIM-PADS '19, June 3–5, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6723-3/19/06…$15.00
https://doi.org/10.1145/3316480.3322896

which is able to adaptively overlap all communication and computation in a given application. In PDES, this is able to very effectively hide overheads of the fine-grained communication [10]. Because the communication is handled by the runtime system rather than the application, in this case Charades, components can be designed modularly and all communication and computation of each component is scheduled by the runtime system. This is relevant to the work in this paper in particular, because the GVT Manager in Charades is a completely separate component from the simulators scheduler and LPs. Each GVT algorithm is implemented as a subclass of the main GVT manager, which allows for easy development and testing of new algorithms. The GVT algorithm proposed in this paper is implemented in this fashion, as are the other two algorithms we compare it against: the Blocking algorithm and the Phase-Based algorithm. The main simulator scheduler interacts with the GVT Manager to determine when the GVT computations should occur, but the computation itself is entirely encapsulated within the GVT Managers, and the communication for the GVT computation is automatically overlapped with other simulator communication by the runtime.

2.2 GVT

In the past, a large number of GVT algorithms have been proposed with a wide range of characteristics. These algorithms can often be categorized by how they affect simulation behavior. Perhaps the most straightforward method for computing the GVT is to halt all simulation progress and wait until there are no events in transit. While this may incur a high degree of overhead, it also computes the exact GVT and has been used at large scales by the Rensselaer Optimistic Simulation System (ROSS) [2, 3, 7]. Because of its simplicity, and success in a large-scale simulator, this is one of the algorithms we will compare against in the results section of this paper.

Algorithms get more complicated once they attempt to reduce overhead by allowing event execution to continue during the computation of the GVT. The SPEEDES simulation system demonstrated an effective and scalable solution to computing the GVT while still allowing event execution to continue [18]. However, in order to take events in transit into account, it still blocks event communication in order to flush the network.

A number of algorithms have also been studied which do not inhibit simulation progress in any way. Gomes et al. [6] and Fujimoto [5] showed effective solutions for computing the GVT in shared-memory simulators. In terms of distributed algorithms, Chen et al. [4] and the ROSS team [1] both developed non-blocking distributed GVT algorithms, however they required either atomic operations or machine clocks. Similarly, Srinivasan developed a non-blocking algorithm requiring specific hardware support for global communication [16]. Our aim is to develop a more general purpose GVT algorithm. One such algorithm is the distributed snapshot algorithm proposed by Mattern [9], and implemented by Perumalla [14] and Mikida [11]. As such, this is one of the algorithms we will compare against in the results section of this paper. The pGVT algorithm is also a non-blocking distributed GVT algorithm, however the GVT management is centralized to a single GVT manager, which receives updates from each LP [8]. It also requires the use of event acknowledgment messages in order to track in-flight events. The Target Virtual Time (TVT) algorithm attempts to calculate the GVT at specific local virtual times, which is a characteristic similar to our Adaptive Bucketed Algorithm [19]. The TVT, however, requires some extra state per event to manage the algorithms vector clocks.

2.2.1 Blocking GVT Algorithm. The Blocking GVT Algorithm we use in this paper is originally based on the GVT implementation used in ROSS [2, 3, 7]. A similar version, which is the version we will use in this paper was also implemented in Charades [11]. The crux of the algorithm is that it blocks event communication and computation until all events have been received at their destinations. Once all events have been received, then there are no events in flight so the GVT is simply the smallest timestamp among all unexecuted events. In the ROSS implementation, the algorithm uses repeated reductions interspersed with network polling to wait for the count of sent events and received events to be equal. In Charades, a special Quiescence Detection library in the Charm++ runtime system is used to monitor events in flight [15]. It maintains counts of events sent and received, but propagates information about these counts during times where the processors are idle. This algorithm is simple to implement, and is able to take every event into account and therefore compute an exact GVT. However, it can incur high overheads by blocking the entire simulation while it waits on events to be received.

2.2.2 Phase-Based GVT Algorithm. Mattern's Phase-Based GVT algorithm is able to compute an estimate of the GVT without blocking event communication or computation [9]. It runs alongside regular event execution in alternating phases, sometimes referred to as white and red phases. When executing in the white phase, all outgoing events are tagged as white events. Similarly, when running in the red phase all outgoing events are tagged as red. As events are sent and received, the algorithm maintains counters for each phase. At some point, determined by the simulator, the GVT will switch from one phase to the other. Now, the number of sent events for the previous phase will not change, since outgoing events are tagged based on the current phase. At this point, the simulator performs a series of repeated reductions to determine when all events from the previous phase have been received. During this time, it must also keep track of the smallest timestamp on any outgoing event. Once it detects completion from the previous phase, it can estimate the GVT by taking the minimum time on each process, including the times of the minimum sent events since the may still be in flight. This algorithm was implemented by Perumalla [13, 14] in an MPI based simulator, and by Mikida [11] in Charades. In Perumalla's implementation, communication was handled via custom implemented MPI reductions. In Charades, which is the version used in this paper, communication was handled by the Charm++ runtime system.

2.3 Models

In this paper we use three different models to evaluate each GVT algorithm. Each of these models were previously used to evaluate the effectiveness of the Charades implementation of Mattern's algorithm [11].

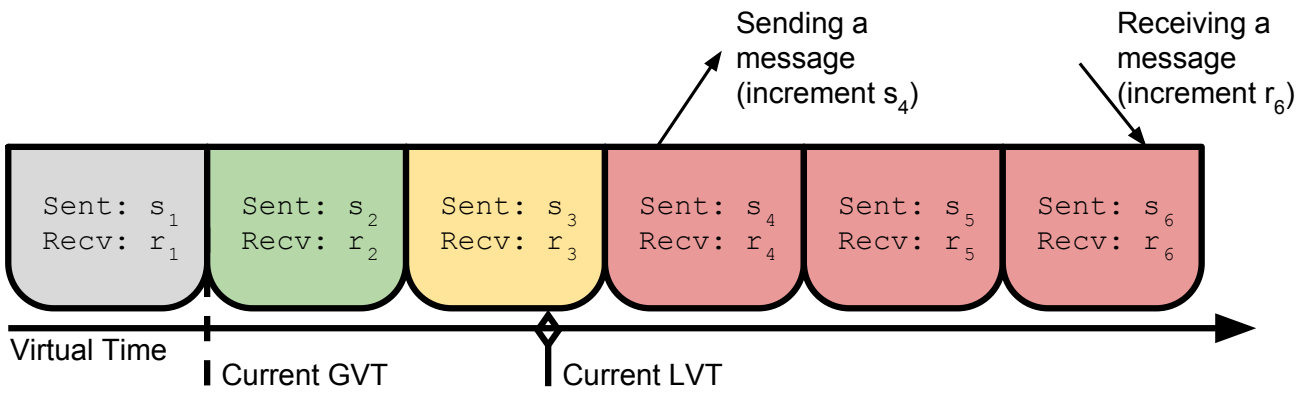

Figure 1: Pictorial representation of Adaptive Bucket GVT.

PHOLD is a common benchmark used to test PDES simulators and involves LPs sending events to either themselves or other LPs chosen uniformly at random based on the model configuration. In order to test slightly more complex workloads the version of PHOLD that we use also allows for imbalance in the amount of time spent executing events as well as the distribution of events across LPs. We test 4 different configurations of PHOLD: Base, Work, Event, and Combo. Base is the standard PDES, where all events are the same weight (take approximately 1 nanosecond to execute), and every LP sends to other LPs with the same probability. In Work and Combo, 10% of the LPs take 10× longer to process and event than others. In Event and Combo, 10% of the LPs are twice as likely to send events to themselves as other LPs.

Dragonfly is a model which simulates communication of packets communicated between nodes in the Dragonfly network topology. The model was originally adapted from the one used in ROSS [10, 12]. In this work we run configurations with four different network traffic patterns. In Uniform, each MPI process being simulated sends packets to another MPI process chosen uniformly at random. In Worst, each MPI process sends all traffic to an MPI process in the neighboring group. Transpose sends to diagonally opposite MPI processes, and Nearest Neighbor sends to an MPI process on the same router.

Traffic is a very basic traffic simulation which consists of cars moving through a grid of intersections to get from a randomly generated source, to a randomly generated destination [11]. We will use four different configurations of this model, each of which consists of a different distribution of sources and destinations. In the Base configuration, all cars choose destinations uniformly at random. In Source and Route, 10% of cars choose their source from a small group of intersections in the top left of the grid. In Destination and Route, 10% of cars choose their destinations from a small group of intersections in the bottom right of the grid.

3 ALGORITHM

The Adaptive Bucketed GVT algorithm is based on the idea of completion detection by counting events, similar to both the Blocking and Phase-Based algorithms we will be evaluating it against. However, it attempts to improve upon these algorithms in a few key ways. It does completion detection based on buckets of virtual time without blocking computation or communication progress of the rest of the simulation. Based on simulation conditions it adaptively determines how many buckets are included in each new GVT estimate. By bucketing events based on their virtual time, the algorithm inherently becomes virtual time aware. The time stamps of events which are being sent and received inform the algorithm on how to proceed, and ultimately what the new GVT estimate will be upon completion of each run of the algorithm.

Figure 1 shows how a single processor splits up virtual time into buckets. In this diagram, the current bucket based on the current Local Virtual Time (LVT) of the processor is colored in yellow. The LVT is simply the smallest timestamp of all unhandled events on the processor. The green bucket represents a bucket which has been passed by the LVT but not the GVT, so in theory we could end up in a situation where rollbacks lower the LVT enough that a green bucket becomes the current bucket again. The grey bucket represents a bucket which has been passed by the GVT and therefore we will never rollback to this bucket or send and/or receive events with timestamps that fall within this bucket. The algorithm also enforces that the GVT will always be at a bucket boundary. Red buckets are future buckets which the processor has not yet reached. As shown in the diagram, each bucket maintains a count of events sent and received, and when an event is sent, the bucket which contains the timestamp the event is scheduled for will increment its sent counter. Because an event can never be scheduled with a timestamp offset less than or equal to zero, this will always be either the current bucket or a future bucket. When an event is received, the bucket which contains its timestamp will increment its received counter. It is important to note that bucket counters are incremented *AFTER* any causality violations are resolved, so if the received event causes a rollback to a previous bucket, the LVT will rollback before counters. Because of this, the received counter that is incremented will always be in the current bucket or a future bucket.

The full algorithm for how these buckets are used to compute the GVT is laid out in detail in Algorithm 1. As events are sent and received on each processor, or when the Scheduler calls gvt_begin, the GVT Manager will check to see if it has passed any buckets beyond the current GVT. If it has, it will contribute the number of buckets passed to an asynchronous All-Reduce (line 10). Once all processors have contributed, each GVT Manager will know the minimum number of buckets passed by all processors, and can contribute its sent and received counters for the buckets in question to an All-Reduce (line 17). Because the simulation has continued during this time, each processor may have advanced past more buckets, or rolled back to include less buckets, so the number of buckets passed is also contributed to the All-Reduce. Upon receiving the All-Reduce, we can check for completed buckets. A bucket is considered completed if and only if its summed sent counter matches its summed received counter, it has been passed by all processors LVTs, and all preceding buckets are also completed. The number of completed buckets are counted up in lines 21 to 25. Once we know how many buckets have been completed, the GVT is updated to the edge of the latest completed bucket (lines 27 to 29). If there are still more buckets which have been passed by all processors but are not yet completed, we continue with further sum reductions until no such buckets remain (lines 30 to 35).

This algorithm differs from the other algorithms in a couple of important ways. One of the more significant differences is that it runs almost completely independently from the simulators scheduler. The other two algorithms require the scheduler to inform them when a GVT should be computed. For the Adaptive Bucketed GVT, it decides when to compute the GVT based on simulation progress. This allows the GVT to adapt to different models effectively. In the next section we will show that this makes tuning the algorithm much simpler.

Furthermore, the algorithm is virtual time aware by virtue of monitoring the timestamps of all incoming and outgoing events. This gives it an advantage over the Phase-Based algorithm, which is not virtual time aware and simply keeps track of event counts. When the scheduler tells the Phase-Based algorithm to begin a GVT computation, it will do exactly one full computation, regardless of simulation characteristics. If, immediately after beginning a computation, there are significant rollbacks, the Phase-Based algorithm cannot adapt to them, because once it begins checking for completion the counter for sent events must remain fixed. This results in some GVT computations advancing very little in virtual time. On the contrary, the Adaptive Bucketed algorithm never has to fix its count of sent events and can shrink and expand the region of virtual time it is considering when rollbacks occur or the simulation is advancing quickly through time. In the next section we will show that this is able to reduce communication required by the GVT when compared to the Phase-Based algorithm.

Algorithm 1 Adaptive Bucket GVT algorithm (Per processor)

```
1:  function BeginGVT()
2:      scheduler->resume()
3:      AttemptGVT()
4:  end function
5:
6:  function AttemptGVT()
7:      lvt = scheduler->min_time()
8:      buckets = (lvt - current_gvt) / bucket_size
9:      if buckets > 0 then
10:         min_reduction(buckets)
11:     end if
12: end function
13:
14: function MinDone(b)
15:     lvt = scheduler->min_time()
16:     new_buckets = (lvt - current_gvt) / bucket_size
17:     reduction(sent[b],recv[b], new_buckets)
18: end function
19:
20: function ReductionDone(sent[b], recv[b], new_buckets)
21:     for x = 0; x < min(b, new_buckets); x++ do
22:         if sent[x] != recv[x] then
23:             break
24:         end if
25:     end for
26:     completed = x
27:     if completed > 0 then
28:         AdvanceGVT(completed)
29:     end if
30:     if new_buckets - completed > 0 then
31:         residual = new_buckets - completed
32:         lvt = scheduler->min_time()
33:         new_buckets = (lvt - current_gvt) / bucket_size
34:         reduction(sent[residual],recv[residual], new_buckets)
35:     end if
36: end function
37:
38: function SendingEvent(event)
39:     sent[event->ts / bucket_size]++
40:     AttemptGVT()
41: end function
42:
43: function ReceivingEvent(event)
44:     recv[event->ts / bucket_size]++
45:     AttemptGVT()
46: end function
```

4 PERFORMANCE ANALYSIS

In this section we show the results of experiments comparing the Adaptive Bucketed algorithm described in the previous section to the Blocking and Phase-Based algorithms described in Section 2. For the Blocking and Phase-Based algorithms we used the best configurations we could achieve, guided in part by the results demonstrated in [11]. For the Adaptive Bucketed algorithm, the primary tuning parameter was the size (in virtual time units) of each bucket. First we will discuss the effects of bucket size on our algorithm. Then we will compare the best configuration of each algorithm on each model configuration described in Section 2. All of these runs are done on Blue Waters, a Cray XE machine at NCSA, using 64 nodes with 32 processes running on each node. Then we will look at

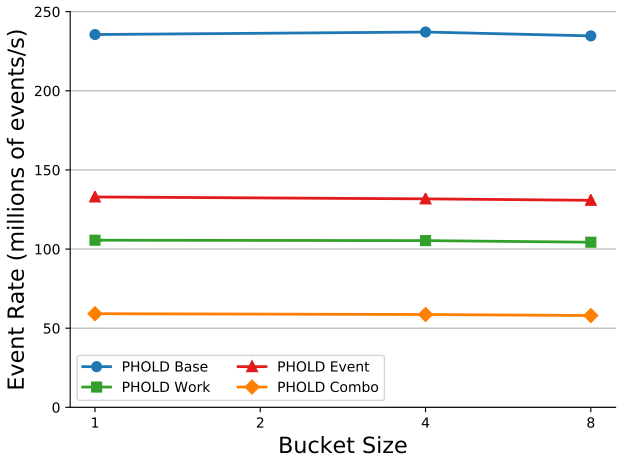

Figure 2: Plot showing how bucket size affects event rate for PHOLD model configurations. Event rate differs by less than 1%.

	Bucket Size		
	1	4	8
PHOLD Base	1013 (1.01)	256 (1.00)	128 (1.00)
PHOLD Work	1021 (1.00)	256 (1.00)	128 (1.00)
PHOLD Event	1016 (1.00)	256 (1.00)	128 (1.00)
PHOLD Combo	1024 (1.00)	256 (1.00)	128 (1.00)
DFly Uniform	773 (10.59)	734 (2.79)	454 (2.25)
DFly Trans	2707 (3.02)	1818 (1.12)	1015 (1.00)
DFly Worst	678 (12.08)	642 (3.19)	646 (1.58)
DFly Neighbor	99 (82.74)	90 (22.75)	81 (12.64)
Traffic Base	806 (1.27)	247 (1.03)	127 (1.00)
Traffic Src	1022 (1.00)	256 (1.00)	128 (1.00)
Traffic Dest	860 (1.19)	255 (1.00)	128 (1.00)
Traffic Route	1019 (1.00)	256 (1.00)	128 (1.00)

Table 1: Table showing the number of completed GVT computations for each of three different bucket sizes. The number in parenthesis shows the average number of buckets included in each GVT.

strong scaling behavior by running the Adaptive Bucketed algorithm on up to 1,024 nodes on Vesta, the IBM BlueGene/Q machine at Argonne National Laboratory. These runs use 64 processes per node, for up to 65, 536 total processes.

4.1 Bucket Size Analysis

While running experiments for both the Blocking and Phase-Based algorithms to compare against, significant effort was required to tune each algorithm. For the Blocking algorithm, changing the frequency of the GVT computation had a significant impact on the resulting event rate of the simulation by effecting both the event efficiency and amount of time spent blocking. For the Phase-Based algorithm, the tuning was primarily required to prevent the simulation from running out of event memory for certain model configurations. Since the Phase-Based algorithm runs without blocking the simulation, the GVT had to be computed frequently so that events could be committed and reclaimed at regular intervals. However, for some of the more complex configurations with low event efficiency, the Phase-Based algorithm would sometimes result in very small advances in virtual time. This resulted in too few events to be committed, and the simulation would crash.

However, this problem did not arise with the Adaptive Bucketed algorithm. Because it is virtual time aware, and progress of the GVT is based on progress of the simulation, the GVT would advance at much more regular intervals. In cases where rollbacks would cause the Phase-Based algorithm to advance the GVT by only a small amount, the Adaptive Bucketed algorithm would adapt the amount of buckets included in the GVT or cancel the computation entirely and wait for a more opportune time to compute the GVT. In cases where problematic rollbacks did not occur, then the Adaptive Bucketed algorithm was able to expand the range of buckets it considered for a given GVT computation. This adaptivity also means that choosing different bucket sizes tends to have very little effect on the overall performance of a simulation. This is demonstrated in

	Bucket Size		
	1	4	8
PHOLD Base	2005 (1.98)	512 (2.00)	260 (2.03)
PHOLD Work	2024 (1.98)	512 (2.00)	258 (2.02)
PHOLD Event	2011 (1.98)	513 (2.00)	257 (2.01)
PHOLD Combo	2040 (1.99)	512 (2.00)	256 (2.00)
DFly Uniform	774 (1.00)	784 (1.06)	782 (1.72)
DFly Trans	2724 (1.00)	2686 (1.47)	1938 (1.90)
DFly Worst	681 (1.00)	656 (1.02)	670 (1.03)
DFly Neighbor	100 (1.01)	91 (1.01)	82 (1.01)
Traffic Base	1276 (1.58)	449 (1.82)	251 (1.98)
Traffic Src	1965 (1.92)	512 (2.00)	256 (2.00)
Traffic Dest	1350 (1.57)	507 (1.99)	262 (2.05)
Traffic Route	2027 (1.99)	512 (2.00)	256 (2.00)

Table 2: Table showing the total number of reductions done by the GVT algorithm for each of three different bucket sizes. The number in parenthesis is the average number of reductions per completed GVT.

Figure 2, which shows the event rate of each PHOLD configuration under three different bucket sizes. The difference in event rate for each configuration is less than 1%, and similar trends hold for the other models as well.

Table 1 shows the total number of completed GVT computations for each model configuration for the three different bucket sizes. In this case, we measure a completed GVT computation as the number of times the GVT manager advances the GVT. It should be noted however, that due to residual buckets in the computation, the GVT Manager may continue doing work even after advancing the GVT. Because of this, the number of GVT computations does not necessarily correspond to the number of times a GVT computation is started, nor does it necessarily correspond directly to the number

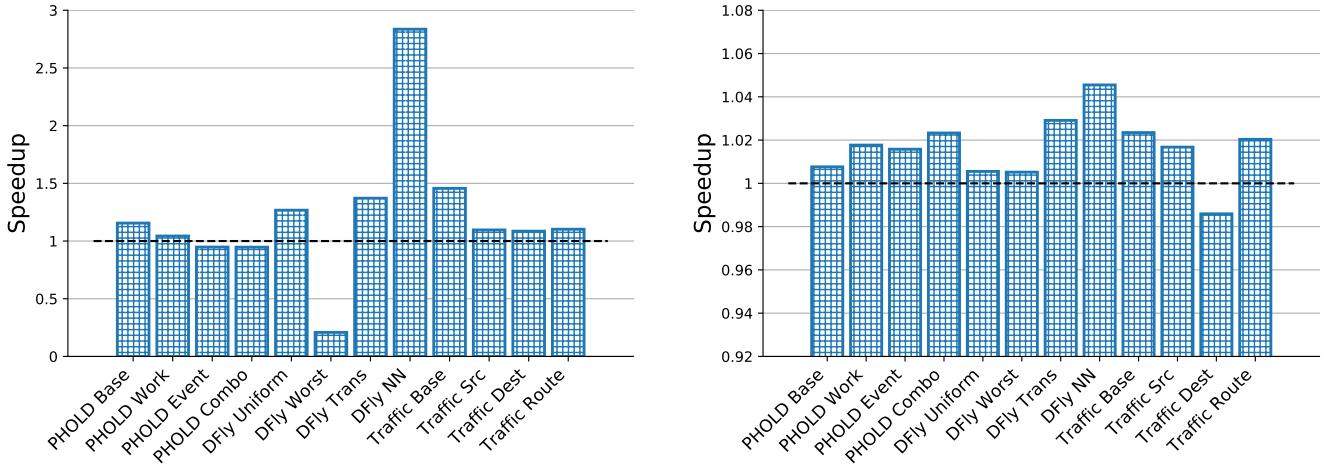

Figure 3: Speedup of the Adaptive Bucket GVT over the best blocking configuration (left) and the best Phase-Based GVT configuration (right).

of reductions required. Table 1 also shows the average number of buckets included in each GVT computation. In many cases we see approximately one bucket per computation. For PHOLD and Traffic, which run a total of 1,024 units of virtual time we see the adaptivity agglomerate some buckets at the smaller bucket sizes. For Dragonfly, the models run for 8,192 units of virtual time, and the GVT counts reveal that many buckets are passed per GVT computation. The Dragonfly model has a lower density of remote communication per unit of virtual time than the other two models, which means the GVT Manager updates with less frequency and more buckets get included in each computation.

The total number of completed GVT computations is only one factor affected by the bucket size, and only reveals part of the picture when considering how much work is being taken up by the GVT computation. When looking at the algorithm, we notice that each GVT computation itself requires a variable number of All-Reduce calls based on how quickly the bucket counts converge. At first glance, it appears that each full GVT call would require at least two reductions. One which starts the GVT computation at Algorithm 1 line 10, and then the subsequent reduction on line 17 where the counts for each relevant bucket are summed and checked. Additionally, the second reduction may have to be repeated multiple times before convergence is reached. However, in practice on the models tested we actually tend to see at most two reductions per GVT, and in many cases even fewer. This is shown in Table 2, which shows both total reductions, and reductions per completed GVT computation. The reason that Traffic and Dragonfly require so few reductions is due to the adaptive portion of the algorithm which constantly updates the number of buckets included in each reduction. When performing the second type of reduction, some buckets are completed, with other buckets still being checked and new ones being pulled into the reduction. Because of this, one reduction to start the GVT computation often results in multiple completed GVT computations. On the other hand the Phase-Based algorithm required on average more than four reductions per GVT

	Blocking	Phased	Bucketed
PHOLD Base	97%	95%	96%
PHOLD Work	76%	52%	53%
PHOLD Event	84%	60%	61%
PHOLD Combo	92%	31%	31%
DFly Uniform	73%	36%	36%
DFly Trans	78%	27%	28%
DFly Worst	91%	2 %	2%
DFly Neighbor	77%	67%	68%
Traffic Base	93%	55%	55%
Traffic Src	93%	16%	16%
Traffic Dest	93%	52%	53%
Traffic Route	95%	15%	15%

Table 3: Table comparing the event efficiency of each model configuration for the best configuration of each of the three different GVT algorithms.

computation in all experiments we ran, and used a similar number of GVT computations as the Adaptive Bucket algorithm with a bucket size of one. This results in a fairly dramatic decrease in the amount of communication required by the Adaptive Bucket algorithm when compared to the Phase-Based one. As previously mentioned this is due to the fact that it chooses to perform the computation based on the progress of the simulation and can adaptively adjust the virtual time span it is considering as the algorithm runs.

Figure 3 shows the event rate of the best Adaptive Bucketed algorithm configuration plotted as speedup over the best blocking configuration (left), and the best Phase-Based configuration (right). In each simulation except for Traffic Dest we see slight speedups over the Phase-Based algorithm, and correspondingly the speedups we see over the blocking algorithm are similar to those demonstrated for the Phase-Based algorithm in [11]. This means that in all but two PHOLD configurations and one Traffic

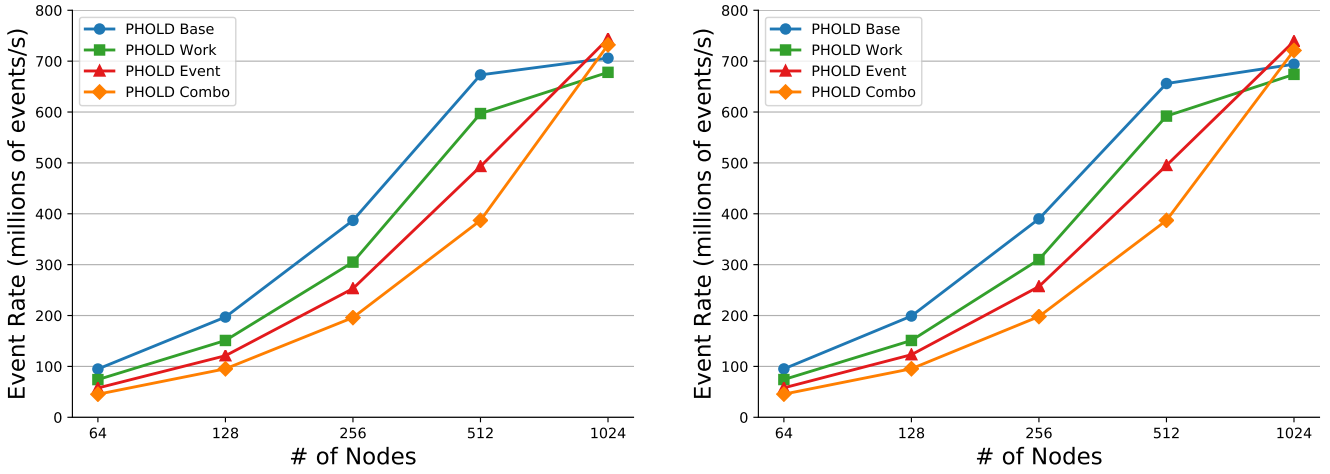

Figure 4: Strong scaling of the Adaptive Bucket GVT algorithm with bucket sizes of one (left) and four (right).

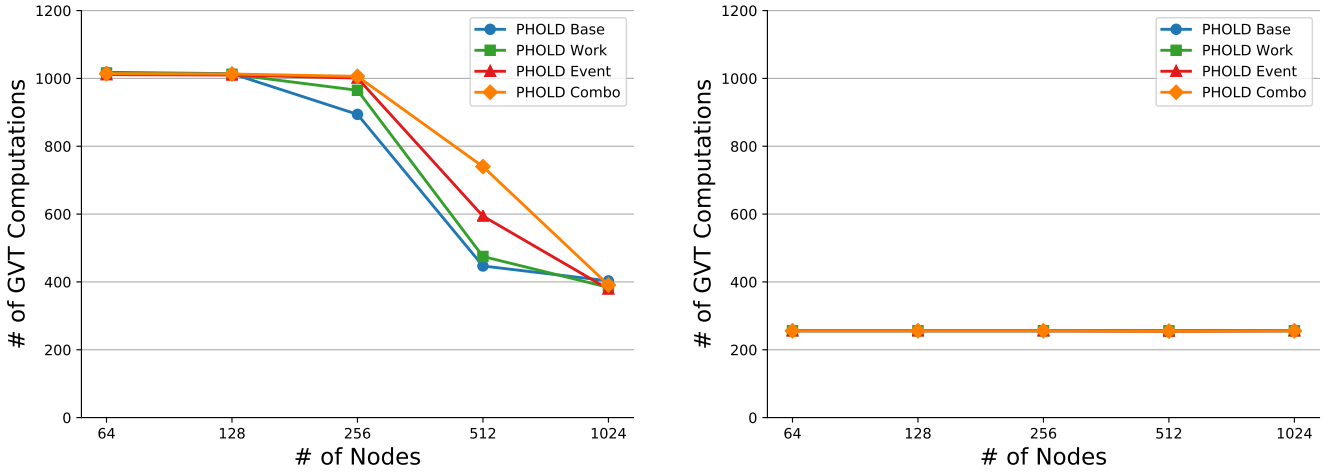

Figure 5: The number of completed GVT computations for each PHOLD configuration at different node counts for bucket sizes of one (left) and four (right).

configuration, the adaptive bucketed GVT algorithm provides the best performance. Table 3 shows what the event efficiency of each model configuration is under each GVT algorithm. There is a clear difference between the Blocking algorithm, and two non-blocking algorithms. In computing the GVT, the blocking algorithm halts execution of events which inherently limits the potential optimism of the simulation. This prevents LPs from diverging widely in timestamp and keeps event efficiency of the simulation relatively high. For both the Phase-Based algorithm and the Adaptive Bucketed algorithm, there is nothing blocking execution and the optimism of the simulation is not limited in any way. Because of this, LPs may end up diverging in virtual time based on model characteristics and how LPs are mapped to processors. This results in both of the non-blocking GVT algorithms having nearly identical event efficiencies. The improvements we see in event rate over the blocking algorithm are therefore due to the reducing the synchronization cost of the GVT since we no longer require event execution to halt while the GVT is being computed. In all cases but DFly Worst, which has extremely low event efficiency when optimism is unbounded, this tradeoff more than makes up for the decreased event efficiency. Similarly, the speedups over the Phase-Based algorithm come entirely from the lowered communication costs of the Adaptive Bucketed algorithm.

4.2 Scaling

Figure 4 shows the strong scaling event rates of the PHOLD benchmark for bucket sizes of one and four. The simulation shows favorable scaling up to 1,024 BG/Q nodes for each configuration. For PHOLD Combo, which is the most complex of the different PHOLD configurations, we achieve 16.17× speedup when comparing the

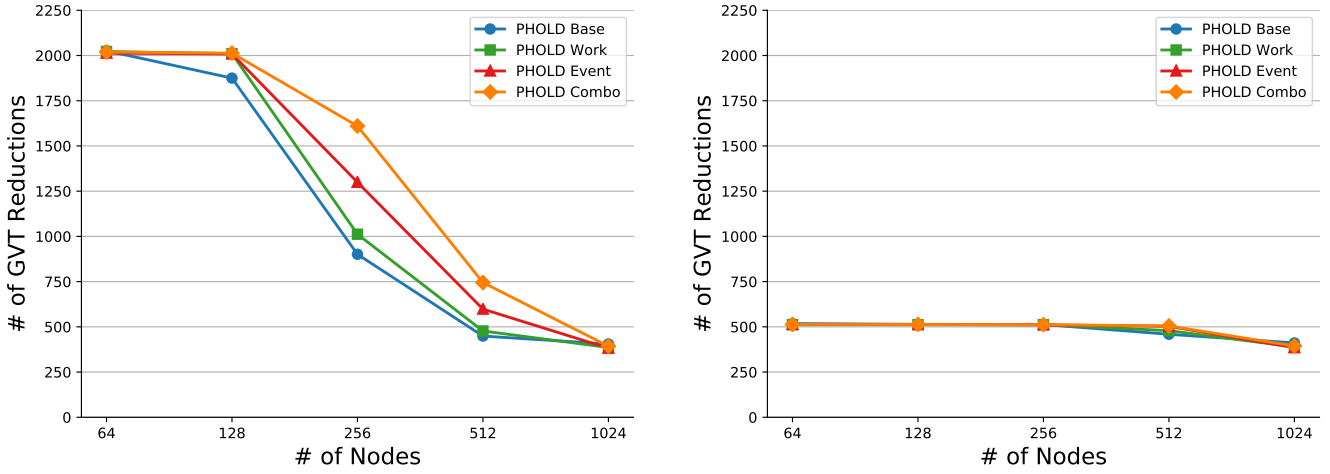

Figure 6: The number of All-Reduce calls for each PHOLD configuration at different node counts for bucket sizes of one (left) and four (right).

1,024 node configuration to the 64 node configuration when using a bucket size of one. It also has a higher event rate for each model at almost every node count than the other two algorithms. For the bucket size of four, scaling is very similar but slightly worse. This is due to the fact that the adaptive portion of the algorithm has more flexibility to agglomerate buckets as the number of nodes changes if the bucket size is kept small.

As with the Phase-based algorithm, the amount of time event execution is blocked by the GVT algorithm is essentially zero at all node counts. The only factor in which the GVT algorithm competes with event execution is contention for communication resources. Figure 5 shows how the number of completed GVT computations scales as we increase node count for the two different bucket sizes. For a bucket size of one, the number of computations changes a lot more with node count. As the number of nodes increases, the number of LPs and events per processor decreases, resulting in fewer events per bucket per process. This means each process moves through buckets faster at higher node counts. With the smaller bucket size of one, the GVT algorithm ends up agglomerating more buckets into a single GVT at higher node counts than it does for the larger bucket size of 4. This also directly results in less communication as shown in Figure 6, which shows the total number of All-Reductions done by the GVT Manager for each node count. This difference results in slightly better scaling for the smallest bucket size, and better performance on the larger node counts. However, the large buckets perform slightly better on the smaller node counts. As discussed previously however, the difference in performance is extremely small. It may be possible to make up for some of that difference by checking for GVT progress less frequently on lower node counts. This would allow for more aggressive agglomeration of buckets, resulting in fewer GVT computations and fewer All-Reduce calls.

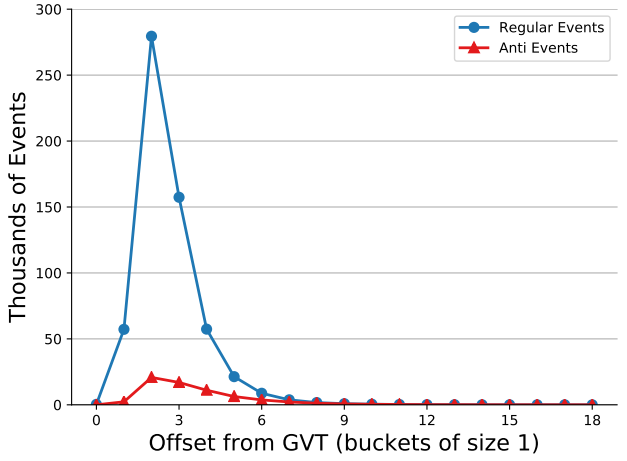

Figure 7: Plot showing the number of regular events and anti events at each offset from the current GVT for PHOLD Combo with a bucket size of one.

5 ADAPTIVE EVENT CONTROL

One major downside of both the non-blocking algorithms we have evaluated is the sharp decrease in event efficiency when compared to the blocking algorithm. The blocking algorithm bounded the optimistic execution of each model as a side-effect of halting event execution to compute the GVT. In this section, we devise a method for improving the event efficiency of the Adaptive Bucketed GVT algorithm without blocking event execution or event communication for those events which are part of the final simulation result.

Our method, inspired by observations from the blocking algorithm as well as ideas used in the SPEEDES system developed by Steinman [17, 18], revolves around the fact that rollbacks are the result processors getting too far ahead in virtual time compared

to others. It is also motivated by the fact that anti events can be far more expensive that local rollbacks. Not only do they incur the added cost of network communication on top of the rollback cost, but they may also lag behind event execution and cause cascading rollbacks. The idea is to allow processors to execute local events unimpeded, but to limit outgoing events that the simulator thinks are likely to be rolled back. Importantly, we should also have minimal impact on the communication of events that will not be rolled back. This should have two effects: higher event efficiency due to the fact that the held remote events will not be executed and rolled back, and fewer messages sent due to a decrease in both regular and anti events that are sent between processors. SPEEDES uses a similar approach in order to implement an asynchronous GVT algorithm [18]. There are a few key differences with our approach however. First of all, the purpose of holding back remote events in SPEEDES is to ensure that the network can eventually be flushed of all in-transit events so that the GVT algorithm can compute the next GVT. In our approach, the GVT does not require that communication is stopped to work properly. The GVT can still be estimated in the presence of events in transit. Here, our purpose for holding back events is solely to improve event efficiency, communication load, and performance. Secondly, in SPEEDES because the GVT requires no in-transit events, once event sending is halted it does not resume until the next GVT cycle. Our approach is able to selectively hold back events based on the risk estimated for each event independently. It may stop certain events, while not stopping others.

In order to limit outgoing events we use the fact that events are already passed through the GVT Manager before they are sent in order to correctly update bucket counts. With only minor changes, we can use this data for more than just computing the next GVT. As events pass through the GVT Manager, the GVT Manager can use the data it has already collected to predict whether or not the event is likely to be rolled back later on. If it thinks the event is likely to be rolled back, then it can temporarily hold the event until some later point in time. To the rest of the simulator, it appears as though this event was sent. If later on it is determined the event must be canceled, a corresponding anti event will be sent. Just like all other events, this anti event will also pass through the GVT Manager. If the GVT Manager is still holding the original event, then the event and anti event can immediately annihilate one another without any network communication. Otherwise, the anti event will be sent as normal. In this way, the GVT Manager can reduce unnecessary communication and lower the risk of cascading rollbacks in a way which is completely transparent to the rest of the simulator.

In order for this to work correctly, the GVT Manager needs to be able to do two things effectively. First, it needs to be able to reasonably predict which events are likely to be canceled so it can hold them back. If it is too conservative in this estimate and lets most events get sent normally, there will be very little benefit to be seen. If it is too greedy, and holds back a large number of events, it may slow down or perturb the rest of the simulation by holding back events which should actually be sent and executed. This may actually hurt event efficiency if other processors regularly have to rollback due to events arriving late. The second thing it needs to be able to do is to decide how long to hold each event. If holding events too long, we may run into the same issue where event efficiency is actually lowered. The longer a correct event is held, the more likely it is that its destination processor will get ahead of it and have to rollback when it is eventually delivered.

	Bucket Size		
	1	4	8
PHOLD Base	0.12	0.03	0.12
PHOLD Work	0.17	0.04	0.02
PHOLD Event	0.18	0.04	0.02
PHOLD Combo	0.16	0.04	0.02
DFly Uniform	0.36	0.09	0.04
DFly Trans	0.64	0.16	0.08
DFly Worst	3.18	0.73	0.39
DFly Neighbor	12.30	2.95	1.75
Traffic Base	0.26	0.07	0.03
Traffic Src	0.32	0.08	0.04
Traffic Dest	1.67	0.42	0.21
Traffic Route	0.31	0.08	0.04

Table 4: Average lag between event and anti event (in buckets)

In this work, we will focus on a fairly simplistic metric for determining which events to hold, and how long to hold them. Again it is heavily influenced by the observation that allowing processors to get far ahead of the most recent GVT tends to lower event efficiency. As such, we will use bucket offset from the current GVT to determine whether or not an event should be held back. Events that are sent very close to the current GVT will be less likely to be rolled back, whereas those events which are sent out many buckets ahead of the current GVT have a higher chance of other events arriving before the event that generated them, and therefore being canceled. In order to back up this assumption, we have added tracing to the GVT Manager which monitors events and anti events based on their offset from the GVT at the time they are sent. Figure 7 shows the results of this tracing for PHOLD Combo with a bucket size of one. We see as we get further from the current GVT, the number of regular events sent decreases at a much faster rate than the number of anti events. The ratio of anti events to regular events increases as the offset increases to the point where every single regular event has an anti event at the furthest offsets. This holds for every other model we have looked at as well. For experiments later in this section, we provide an offset cutoff to determine when to start holding events. If an event has an offset higher than the cutoff, it is held. Otherwise the event is sent as normal. These offsets can be estimated by looking at the aforementioned traces, and tuned over repeated runs.

In order to hold back events based on offset from GVT, the GVT Manager maintains an additional set of buckets. These buckets are based on offset from GVT and hold hash tables of events rather than event counts. When an event is sent, if it is an event that should be held because its offset is greater than the cutoff, it is hashed in the bucket corresponding to its offset. For example, if an event is sent with a timestamp that is four buckets away from the current GVT it will be hashed in the fourth offset bucket. One important thing to note is that since this is based on the events offset when

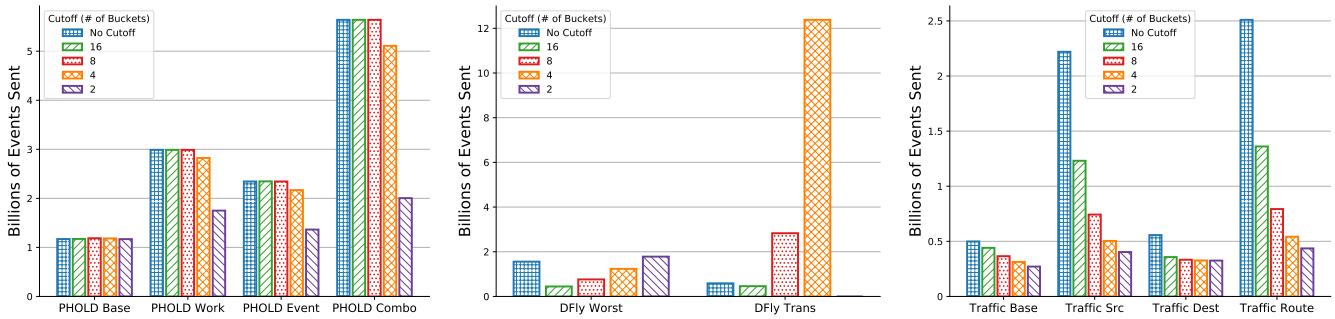

Figure 8: Figure showing the number of events sent remotely for each model when holding back events based on their offset from the current GVT. The higher the offset cutoff, the further from the GVT an event has to be to be held. Therefore bars are arranged by increasing aggressiveness.

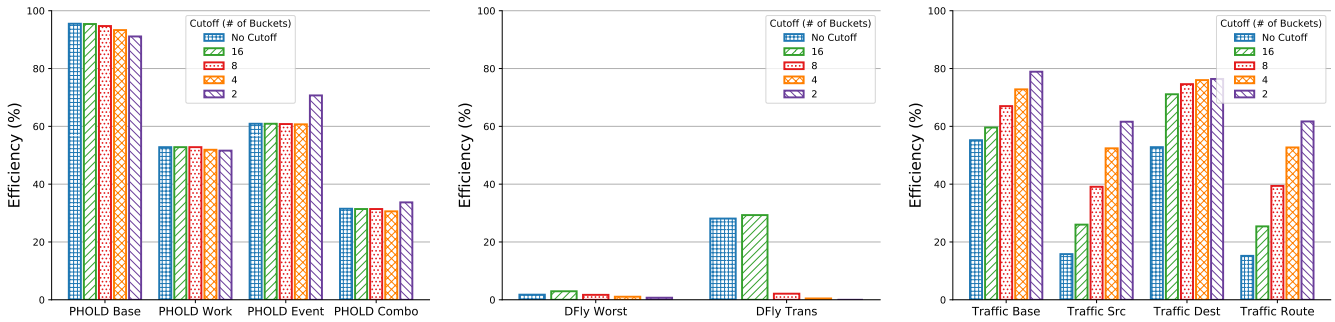

Figure 9: Figure showing the efficiency for each model when holding back events based on their offset from the current GVT.

sent, events in the same hash bucket may not belong to the same absolute time bucket. The offset of the event is also stored in the event itself. That way, when an anti event is sent, the GVT Manager can get the offset of the original event and check if that event still exists in the corresponding offset bucket. If it does, the event is canceled and no communication is needed. If not, the anti event is sent as normal.

As the simulation makes progress, events which are held back eventually need to be released. Since progress is determined by the GVT advancing, we also use this as a time to release held events. After a GVT is completed and the GVT has advanced some number of buckets, we can look at held events and determine which ones to release. At the very least, held events which are now in the current bucket must be released otherwise the GVT cannot properly progress. In order to determine which events to release, we also rely on data collected by the GVT Manager to get some intuition about how cancellations work. Table 4 shows the average amount of lag between event send and event cancellation for each model configuration. This is computed by taking the difference between the offset at which the original event was sent and the offset of the corresponding anti event. This is the same as the number of buckets which are completed by the GVT algorithm by the time the anti event is sent. These numbers can be used to determine how many GVT buckets an event should be held for before being released. For almost all models in the table, we see numbers less than one. This means that, on average, anti events are sent during the same GVT bucket as their corresponding regular events. Because of this, for most models we release all events the next time a GVT is computed. As stated previous, holding events too long may hurt performance by causing rollbacks when we eventually do release them and they reach their destinations late.

First, Figure 8 shows the effects on the event communication by showing the total number of events (including anti events) sent for each model. For this figure, and all upcoming figures, data for cutoffs of 2, 4, 8, and 16 buckets, as well the base case of no throttling. The bars are arranged from most conservative to most aggressive in terms of number of events held. By correctly holding back an event that would eventually be canceled we save at least two messages: one for the original event and one for the subsequent anti event. It could even save more events if there is a cascading effect where many anti events are required to chase down an incorrect event. This is where we see the most direct, and drastic, effects of throttling. For PHOLD configurations, event communication is cut by 30−55%. DFly Worst, and many of the Traffic configurations have their communication cut by more than 70%. The Dragonfly results also highlight a potential pitfall of holding events too aggressively. As more events are held, the total number of events sent actually decreases. As the next plot will show this is at least partially due to decreased event efficiency.

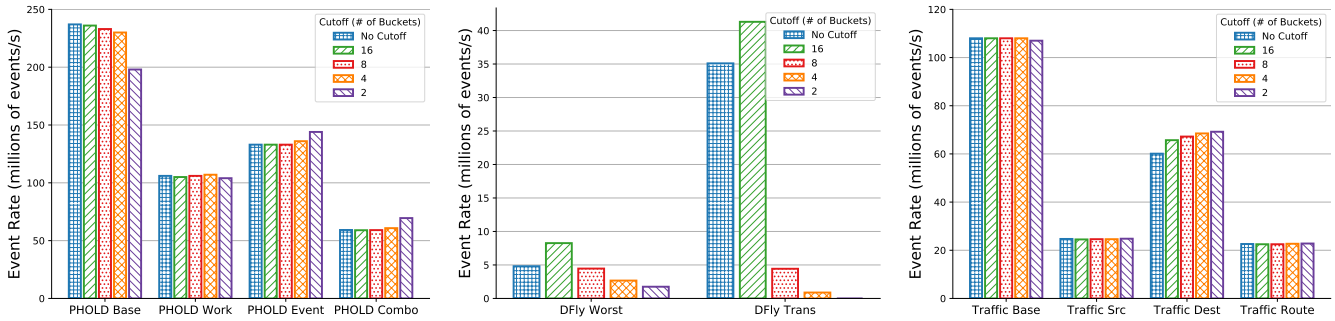

Figure 10: Figure showing the event rates for each model when holding back events based on their offset from the current GVT.

Figure 9 shows the effect on efficiency for each configuration. In many cases these results mirror the effects on total communication, the few exceptions being PHOLD Base and PHOLD Work. PHOLD Base already has exceptional event efficiency so holding back events is only causing delays that hurt efficiency. For Dragonfly, the net gain in event efficiency is small, but relative to starting event efficiency it still equates to significantly fewer rollbacks. For example, the number of rollbacks in DFly Worst are cut down to 60% of what they were with no throttling. For Traffic, we see far more drastic effects. The event efficiency of all model configurations sees a significant increase, with Traffic Source and Traffic Route roughly tripling their event efficiency.

Finally, to see how these changes affect overall performance, we plot event rate of each model in Figure 10. We see varying performance based on configuration. For PHOLD Base, which is extremely uniform and already has a very high efficiency, holding events actually hurts performance. Similar results were seen for DFly Uniform, DFly NN, and Traffic Base. DFly Uniform and NN are omitted to improve chart readability. For each other PHOLD configuration holding events improves event rate by 1% for PHOLD Work up to an 18% improvement for PHOLD Combo. For the two Dragonfly configurations shown, we also see improvement of 70% and 15% for the two Dragonfly models. Both of these models suffered from very low event efficiency. As before, we see that being overly aggressive can ultimately hurt performance. Traffic does not see quite as much improvement as PHOLD or Dragonfly, but in the best case, Traffic Dest sees a 15% improvement in event rate. In the case of other models, it is likely that over aggressive throttling, although improving event efficiency significantly, slowed down overall progress by holding too many events. A scheme more targeted at holding events on inefficient processors while allowing other processors to send uninhibited may be able to address this particular issue.

Overall, depending on the model examined, we are able to improve performance by holding back events that are likely to be canceled in the future. However, we also believe there is significant room for improvement here by choosing more effective ways to select which events to hold. Currently, we set a static cutoff in distance from GVT, and use that cutoff for every processor in the simulation at all times. The analysis was done offline and based on aggregate results across an entire run. It seems feasible that online analysis, and throttling on a per processor basis could even more effectively utilize the data and structure present in the Adaptive Bucketed algorithm.

6 CONCLUSION

In this paper, we have proposed a new GVT algorithm, the Adaptive Bucketed algorithm, which has a few important characteristics. First, it is completely non-blocking, as it allows the computation and communication of other simulator tasks to continue unimpeded. Because of this, it achieves up to 3× speedup for certain models compared to the Blocking algorithm. This comes despite the fact that it runs with significantly lower event efficiency.

Secondly, it is virtual time aware, which allows it to effectively adapt to simulation characteristics by deciding when to begin and how much virtual time to include in the computation. It is even able to shrink or expand its scope while running. This is able to keep to amount of communication required low, especially when compared to the other non-blocking algorithm evaluated for this paper. Due to this fact it also achieves modest speedup over the Phase-Based algorithm, even though both are non-blocking. It also shows very favorable strong scaling, especially as it adapts to different simulation characteristics at different node counts.

Finally, by looking at the data collected by the algorithm and exploiting its bucketed structure, we were able to further improve performance for certain models by selectively holding back events which were predicted to be rolled back later on. This decreased the total amount of events sent across processes and often had a positive effect on event efficiency as well. We also believe this is an important avenue for future work. In the results presented here, analysis was all offline, and each process behaved the same. As future work, we would like to explore various online analysis techniques which would allow processes to individually adapt to evolving simulation characteristics, using the data and structure already inherently present within the Adaptive Bucketed algorithm.

REFERENCES

[1] 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.

[2] 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.

[3] C. D. Carothers and K. S. Perumalla. On deciding between conservative and optimistic approaches on massively parallel platforms. In *Proceedings of the 2010 Winter Simulation Conference*, pages 678–687, Dec 2010.

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

[5] 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.

[6] 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.

[7] A. O. Holder and C. D. Carothers. Analysis of time warp on a 32,768 processor ibm blue gene/l supercomputer. Citeseer.

[8] B. Kannikeswaran, R. Radhakrishnan, P. Frey, P. Alexander, and P. A. Wilsey. Formal specification and verification of the pgvt algorithm. In *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, FME '96, pages 405–424, London, UK, UK, 1996. Springer-Verlag.

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

[10] 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.

[11] E. Mikida and L. Kale. Adaptive methods for irregular parallel discrete event simulation workloads. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '18, pages 189–200, New York, NY, USA, 2018. ACM.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] J. Steinman. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. In *Proceedings of the 1991 SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 95–103, January 1991.

[18] 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.

[19] A. I. Tomlinson and V. K. Garg. An algorithm for minimally latent global virtual time. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, PADS '93, pages 35–42, New York, NY, USA, 1993. ACM.