ADAPTIVE TECHNIQUES FOR SCALABLE OPTIMISTIC PARALLEL DISCRETE EVENT SIMULATION

BY

ERIC MIKIDA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kale, Chair
Professor David Nicol
Professor Marc Snir
Dr. David Jefferson, Lawrence Livermore National Laboratory

ABSTRACT

Discrete Event Simulation (DES) can be an important tool across various domains such as Engineering, Military, Biology, High Performance Computing, and many others. Interacting systems in these domains can be simulated with a high degree of fidelity and accuracy. Furthermore, DES simulations do not rely on a global time step and simulated entities are only updated at discrete points in virtual time at which events occur. The particular DES simulation engine handles simulation logic and event scheduling, while the particular models written by domain experts need only focus on model-specific logic. As models grow in size and complexity, running simulations in parallel becomes an attractive option. However, a number of issues need to be addressed in order to effectively run DES simulations in parallel in a distributed environment. The issue of how to synchronize PDES simulations has been addressed in a number of ways, using various types of either conservative or optimistic protocols. Optimistic simulation synchronization has shown several benefits over conservative synchronization, but it is also more complex and brings with it some unique challenges. Two of these challenges are synchronizing event execution across distributed processes, and maintaining a high accuracy in the speculative execution of events. This thesis aims to address these challenges in order to make optimistic simulations even more effective and reliable. Specifically, this thesis explores a variety of GVT algorithms in an attempt to lower synchronization costs, while utilizing other techniques such as dynamic load balancing to maintain a high event execution efficiency and keep work balanced across execution units. Most importantly, these techniques aim to make the simulator robust and adaptive, allowing it to work effectively for a variety of models with different characteristics and irregularities.

In loving memory of my grandfather, Robert Ott

ACKNOWLEDGMENTS

Getting a Ph.D. is an arduous task, probably one of the hardest things I've done in my life. At many points it seemed impossible to complete and that I had been beaten, but luckily I had many people supporting and helping me along the way. My advisor, Laxmikant "Sanjay" Kale was always there to guide me in the correct direction, and there was never any doubt that he had my best interests in mind. Furthermore, being part of such a large and collaborative group in the Parallel Programming Laboratory meant that there were always people around ready to help. In particular, Nikhil Jain was a great mentor in the early parts of my Ph.D. journey, and set me up for success throughout the process. There was also Ronak Buch, who if it wasn't for him, I may have finished my Ph.D. much earlier but with much less of my sanity remaining. Our time in the office was full of long meandering conversations, sometimes relevant to PPL, computer science in general, or absolutely nothing at all. Either way it certainly made this journey more enjoyable.

Of course, I also wouldn't be here without the support of my family and friends either. My parents set me up for success at a very early age, and supported me throughout my entire academic journey, even when it took me far from home. I would not be where I am today without them. My wife Ariel also took a great leap of faith when she followed me out to Illinois when I decided to pursue my Ph.D., and has been by my side to support me ever since. Because of her, I knew I was never truly alone despite the challenges that were presented to me, and I always had plenty of motivation to continue moving forward. When things felt bleak my brother Cory was able to commiserate with me as someone also going through the process, and when all else failed Andrea Delgado was there any time I needed to let off some steam.

Finally, I have to thank my grandfather, Robert Ott, without whom I may have never even begun this journey in the first place. From a very early age he sparked an appetite for logic and problem solving that still persists to this day.

TABLE OF CONTENTS

CHAPTER 1: OVERVIEW

Parallel Discrete Event Simulation (PDES) can be a powerful and important tool in many relevant domains. However, it can be difficult to coordinate event execution across concurrent processes efficiently. Furthermore, typical PDES models frequently involve large amounts of irregular, dynamic behavior. This irregular behavior can not often be determined *a priori*, which means a PDES engine will need to be able to adapt on the fly to run effectively. Certain PDES algorithms also require speculative execution of events which, can add to the irregularity and make model behavior even more difficult to predict.

1.1 DISCRETE EVENT SIMULATION

A Discrete Event Simulation (DES) consists of *Logical Processes* (LPs) and events. LPs represent the entities within the simulation and encapsulate the majority of simulation state. Changes within DES are driven by execution of events that are scheduled at discrete points in virtual time and executed by the specific LP which they affect. In order to obtain a correct simulation, the events must be executed in timestamp order. The execution of one event may cause one or more events to be scheduled for a later virtual time. Events can never be scheduled at a virtual time that is earlier than that of the scheduling event. Because of this restriction, as long as the scheduler for a sequential DES simulation always picks the event with the smallest timestamp to execute, the events will always be executed in a non-decreasing timestamp order. Therefore, implementation of a sequential DES simulation engine often comes down to simply maintaining a priority queue or other similarly ordered data structure of events. At each iteration, the event with the smallest timestamp is popped from the queue and executed. Newly scheduled events are inserted into the queue, and execution continues until the queue is emptied or some other termination condition is met. An example of such a scheduler is shown in Listing 1.1.

Listing 1.1: Basic sequential simulator

```
1 while (!done) {
2   // pq is a priority queue of events sorted by timestamp
3   Event* e = pq.pop();
4   LP* lp = e->getTargetLP();
5
6   // Execution of the event may result in new events being added to pq
7   lp->execute(e);
8 }
```

As simulations get larger and more complex, running them on sequential machines becomes less feasible. Exploiting parallel machines, whether they be multi-threaded shared memory machines or distributed memory clusters and supercomputers, will allow us to run larger and more detailed simulations and provide faster time to solution for existing models. Running DES simulations in parallel comes with its own set of unique challenges, some of which we address in this thesis. Like sequential DES, PDES simulations consist of LPs and timestamped events, which need to be executed in timestamp order to get the correct final result. One of the primary challenges in a PDES simulation is how to ensure the correct execution order of events while effectively distributing work and data across multiple concurrent processes. For shared memory machines, execution of events must be effectively distributed while avoiding race conditions on simulation state and simulation entity data structures. On distributed memory machines the simulation data is also distributed and events must be communicated across the network. This further complicates the simulators job due to the fact that there is now a real time delay between the time an event is created, and the time it arrives at its destination. Synchronization across processes is required in order to maintain the causal order of event execution. Significant research effort has been invested in order to figure out how to best synchronize PDES simulations. As a result, two main schools of though have emerged: conservative synchronization and optimistic synchronization [1, 2, 3, 4, 5].

1.1.1 Conservative Synchronization

Conservative synchronization requires that no event in a PDES simulation be executed by an LP unless it can be guaranteed that no event with a smaller timestamp will be scheduled for that LP. This ensures that each LP will execute its events in non-decreasing order. Since an LP entirely encapsulates its own state, which is only changed when the LP executes events, the LP will have the same state at the end of the simulation as if the simulation were run sequentially. Conservative simulations must therefore be able to quickly and correctly determine which events are safe for execution without deadlocking. In doing so, the simulator must also make sure to expose enough parallelism to effectively utilize the available execution resources. There are multiple different conservative simulation techniques which aim to accomplish this. The null message approach relies on static FIFO communication channels between communicating processes [4, 5]. A process can determine if it can safely execute a given event by inspecting the message timestamps on incoming channels and determining what the smallest possible timestamp could be for any future incoming events. This approach requires communication of additional "null" messages which

serve no purpose other than to give additional safety guarantees to avoid deadlocks. Another common approach is to determine safe windows of virtual time, like in Nicol's YAWNS protocol [2]. YAWNS uses global synchronization and an additional model parameter called lookahead to determine windows during which no new events will be scheduled. Lookahead is defined to be the smallest amount of virtual time between an event being executed and new events which it creates. The windowed structure of this algorithm shares similarities with algorithms we will be exploring later in this thesis. These algorithms can be simple to implement, and do not require any extra functionality from models. However, they may limit parallelism presented to the simulator, and performance may heavily depend on model characteristics. In [2], Nicol shows gives analytical bounds on the protocols performance, and demonstrates that if communication cost per event is low then performance is asymptotically the same as a serial simulation. However, this is also dependent on the model having a high degree of concurrent activity. In fact, in [6], it is shown that the restrictive time windows of YAWNS may hinder performance, and that increasing the size of the window by introducing optimistic execution can lead to asymptotically better performance. However, YAWNS can still do well when there are many LPs per process, which increases the amount of concurrent activity, by avoiding overheads associated with optimistic protocols.

1.1.2 Optimistic Synchronization

In conservative synchronization, exposing enough parallelism amongst events to effectively utilize parallel resources can be difficult. Even with an efficient and low overhead simulator, parallelism may still be extremely limited just by the characteristics of models being run. At any given virtual time, there will only be so many events that can be safely executed. Optimistic synchronization aims to remove this restriction by allowing processors to more freely execute events, independently of model characteristics and restrictions. It does so by executing events speculatively, and relying on mechanisms for recovering from causality violations should they occur. The TimeWarp protocol developed by David Jefferson [3] is the most common optimistic synchronization algorithm. Upon detecting a causality violation, which occurs when an LP receives an event with a timestamp smaller than its most recently processed event, the LP must *rollback* to a virtual time prior to that of the incoming event. Rolling back an LP to time t is defined as restoring the LPs state to what it was at time t, including the contents of its pending event queue. After a rollback, execution can continue, which will result in re-execution of events in the pending queue. This means that the entire event history of an LP must be saved in case a rollback would require events to be re-executed. Two common mechanisms for actually performing rollbacks are state saving, ie

checkpointing, and reverse computation. State saving relies on creating backups of an LPs state at certain virtual times. Performing a rollback just requires restoring an LPs state to one of the backups. Reverse computation is done by undoing past events in reverse order until an LP reaches the target virtual time. Starting from the most recently executed event, the rolling back LP will undo the effects that forward execution of the event had on its state. It then puts the event back in the pending queue, and repeats the process for the next most recently executed event. This usually requires model writers to write extra code for reversing the effects of events, but also may use less memory than saving entire LP states. Research is also being done which would allow the reverse computations to be compiler generated, therefore not requiring extra code from the model developers [7].

A number of simulators have shown good performance and high scalability using the Time-Warp protocol. Georgia Time Warp (GTW) is a shared memory simulator which utilizes state saving as the rollback mechanism [8]. ROSS, which also served as a starting point for much of the work in this thesis, is a highly scalable distributed simulator which uses reverse computation as the means to rollback events [9]. However, the increased parallelism presented by the TimeWarp protocol comes at the cost of overheads for managing rollbacks and event cancellations, which occur when a rolled back event has already sent a subsequent event to a remote processor. In order to limit these overheads, protocols such as Bounded TimeWarp (BTW) have been proposed, which limit the amount of optimism allowed by the simulator by executing in bounded time windows similar to YAWNS [10, 6]. Breathing TimeWarp is a different approach for limiting optimism by completely removing the possibility for non-local rollbacks (rollbacks caused by receiving an event from another processor), and was used as the synchronization protocol in the SPEEDES simulator [11, 12].

GVT Computation

Due to the fact that these optimistic approaches require extra memory for storing events and states, we also need some way to periodically reclaim this extra memory. More fundamentally, since LPs can be rolled back to older times, how can we ever be sure that the simulation is completed or be certain that cascading rollbacks wont take us back to the beginning of the simulation? The inter-related notions of "committing an event and of Global Virtual Time (GVT) address these concerns. An event is said to be committed when the simulation engine can guarantee that it will not be rolled back. The GVT is the latest virtual timestamp which has been reached by every LP in a simulation. Because LPs can only schedule events for timestamps in their future, the GVT defines a lower bound on the timestamps of events which can be created by the model. Since no events will be scheduled

before the GVT any events previously executed at timestamps smaller than the GVT are guaranteed to not be rolled back. These events can therefore be committed, and any extra state or event memory can be reclaimed. Furthermore, when speculatively executing events, certain operations such as writing to file may be difficult or infeasible to rollback from. To deal with this, these effects are not performed until an event can be committed.

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 will still allow for the simulation to commit events and reclaim memory, albeit less effectively than if the actual GVT was used. A significant amount of research 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. [13] and Fujimoto et al. [14]. On distributed systems, non-blocking algorithms which rely on atomic operations and machine clocks were studied by Chen et al. [15] and as part of the ROSS simulation system in [16]. Srinivasan's implementation in [17] 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 [18]. Mattern and Perumalla have also developed distributed snapshot algorithms which can be used to compute GVT without stopping event execution or communication [19, 20]. We present and analyze a number of GVT algorithms in more detail in Chapter 4.

1.1.3 Synchronization Comparison

There are pros and cons to both conservative and optimistic approaches to PDES synchronization. In conservative synchronization, performance can become heavily model dependent due to limited parallelism. Specifically, models with inherently low lookahead values may have a very limited selection of events that are able to be safely executed in parallel [6]. Conservative synchronization does have a low memory overhead, and does not require any

extra code to be written for committing events or making them reversible. This makes the conservative approach fairly robust and easy to implement. The optimistic approach has much larger potential to exploit parallelism by not restricting processes to only execute safe events. This can result in much higher parallel efficiency and speedup. However, optimistic simulations have much higher memory requirements in order to be able to execute speculatively. Additionally, the GVT computation can incur significant overhead, especially for models which require frequent GVT computations to keep up with memory consumption. Furthermore, there is the added overhead of dealing with causality violations via rollbacks and event cancellation. Simulations that execute incorrect events frequently will end up doing more work than a simulation which only executes safe events. This can counteract the speedup gained from the extra parallelism. This can become especially problematic for simulations that create a high amount of anti-events which have the possibility of cascading out of control.

1.2 DYNAMIC LOAD BALANCING

Dynamic load balancing is a widely studied problem in the field of parallel computing [21, 22, 23, 24]. Achieving a good balance of work across execution units in parallel applications can be an important factor in achieving good performance and scalability. PDES models can have a wide variety of performance characteristics and behaviors, many of which may result in irregular or dynamically changing loads across the system. This can be exacerbated by speculative execution, which can also have a significant effect on the balance of work performed by different parts of a simulation. Load balancing has been shown to be effective at improving performance and efficiency in other domains [25, 26, 27]. Work has also been done to tailor load balancing strategies specifically for PDES simulations, yet it continues to be an area of active research. In [28], load balancing was shown to be effective for PDES applications running in a network of multi-user workstations. Similarly, [29] demonstrates dynamic load balancing on shared-memory processors by utilizing active process migration. Meraji et al. [30] shows benefits of load balancing for gate-level circuit simulations, and Deelman et al. utilizes load balancing for explicitly spatial models [31].

1.3 SUMMARY

The goal of this thesis is to improve upon the optimistic approach to synchronizing PDES by mitigating some of the cons described above. We aim to make improvements to the

robustness of optimistic simulations by focusing on effective GVT algorithms and techniques for improving speculative execution. We also look at techniques to improve the adaptivity of the simulator to allow it to more effectively handle the complexity of speculative execution across a variety of models. In doing so, we hope unlock the full potential of the increased parallelism provided by the optimistic approach.

Many techniques have been developed to improve the effectiveness of PDES simulations with varying degrees of success. The goal of this research is to demonstrate how many of these techniques can benefit from an adaptive message-driven environment, highlight interplay between these techniques, and to develop new techniques to make PDES an even more powerful tool at much larger scales. In this report we will discuss three major pieces of work towards this goal:

- Research and design of a PDES simulator within an asynchronous and message-driven programming model
- Research in the area of non-blocking and adaptive GVT algorithms
- Research in the area of dynamic load balancing in the face of speculative execution

1.3.1 Message-Driven Simulator

In Chapter 3, we discuss the design of Charades, a PDES simulator built on top of the Charm++ parallel runtime system. This is an evolution of an earlier project to create a version of the Rensselaer Optimistic Simulation System (ROSS) on top of Charm++ to demonstrate key benefits of a message-driven programming model on PDES [32]. From the initial Charm++ version of ROSS, much of the underlying simulation framework design was further adapted and changed to better fit the Charm++ programming model. Charades provides the foundation for the rest of our work, but also shows at a very core level some of the benefits implementing PDES in an asynchronous message-driven programming model by improving the performance of models tested in both systems. In particular, we show that message-driven scheduling can effectively hide the latency of the fine-grained messaging in many PDES simulations by dynamically overlapping communication and computation based on the availability of work. In later chapters, the message-driven programming model also allows for effective overlap of distinct simulator components, allowing for modular design of more complex algorithms.

1.3.2 Reducing GVT Synchronization Costs

In Chapter 4, we discuss the GVT framework in Charades and how it interacts and overlaps with the rest of the simulator. We then look at a variety of GVT algorithms implemented in Charades and discuss their overheads and other effects they have on important aspects of simulation behavior. By relying on message-driven execution to overlap GVT work with the rest of the simulation, we can effectively implement non-blocking GVT algorithms which adapt to simulation conditions. We finish the chapter by presenting a new virtual time aware GVT algorithm which operates without blocking event execution. We show its scalability and performance in comparison to existing GVT algorithms, and discuss further improvements which aim to control optimism without incurring more synchronization.

1.3.3 Balancing Speculative Load

In Chapter 5, we look to further improve simulation performance and robustness by introducing dynamic load balancing into the Charades simulator. We discuss the load balancing framework within the Charm++ runtime system and how it interacts with Charades. We then study a number of strategies for balancing load in Charades, and propose a variety of metrics which can be used to attribute load to LPs in a simulation. Each of these metrics captures different aspects of simulation characteristics, and depending on which model we use, different metrics perform better. We conclude the chapter by combining each of the techniques proposed in this thesis, and show that dynamic load balancing is able to further improve the performance of the non-blocking GVT algorithms studied in Chapter 4.

CHAPTER 2: EXPERIMENTAL FRAMEWORK

In this chapter we will briefly describe the framework for our experimental setups. In order to effectively demonstrate the techniques laid out in the later chapters of this thesis, we have developed a simulator built on top of Charm++ and based heavily on wisdom gained by working with the Rensselaer Optimistic Simulation System (ROSS). This framework, along with several models also described in this chapter, gave us the means to research a variety of techniques toward our goal of a more robust and efficient optimistic simulation system.

2.1 CHARM++

Charm++ is a parallel programming framework which consists of an adaptive runtime system that manages migratable objects communicating asynchronously with each other. It is available on all major HPC platforms and takes advantage of native messaging routines when possible for better performance. Applications developed for Charm++ are written primarily in C++ with a small amount of boiler-plate code to inform the runtime system of the main application entities (C++ objects).

Charm++ is much more than just a messaging layer for applications. It contains a powerful runtime system that aims to remove some of the burden of parallel programming from the application developer by taking care of tasks such as location management, adaptive overlap of computation and communication, and object migration. Broadly speaking, the primary attributes of Charm++ that we hope to leverage in PDES are: object-level over-decomposition, one-sided asynchronous messages, message-driven execution, and object migratability.

In Charm++, the application domain is decomposed into work and data units, implemented as C++ objects called chares that perform the computation required by the application. Thus, the programmers are free to write their applications in work units natural to the problem being solved, instead of being forced to design the application in terms of cores or processes. Typically there are many more objects than cores. This over-decomposition gives the runtime more flexibility in scheduling and location management. An application may also have many different kinds of chares for different types of tasks. Figure 2.1 (left) shows a collection of 6 chares of two different types labeled A, B, C, D, E, and F. Arrows represent communication between chares. This freedom of decomposition is highly suitable for implementing PDES engines and models, since they typically have simulated entities of different types whose work and count is determined by the model being simulated.

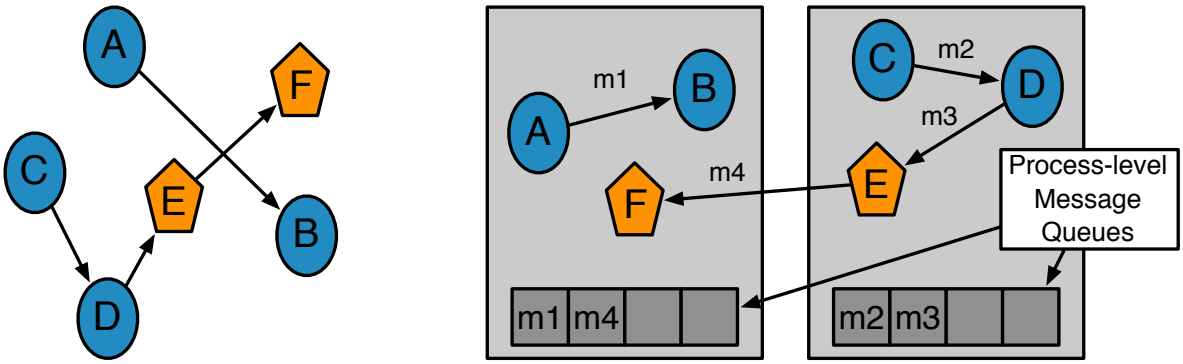

Figure 2.1: A depiction of communicating chares from the programmers view (left) and the runtime system's view after the chares have been mapped to processes (right). Arrows represent messages sent between chares.

The runtime system is responsible for mapping the chares defined by an application to cores and processes on which they are executed. The chares interact with each other asynchronously in a one-sided manner: the senders send messages to recipient objects and continue with their work without waiting for a synchronous response. This mode of communication is a direct match to the type of communication exhibited by a typical PDES simulation, where LPs send events scheduled for other LPs without the recipient LP needing to know about or actively participate in the communication process. Figure 2.1 (right) shows a potential mapping of the 6 chares from the first example to specific hardware resources. The runtime system keeps track of the location of each chare and handles sending messages across the network when communicating chares are mapped to different hardware resources.

The scheduler of the Charm++ runtime system is entirely message-driven. It schedules chares for execution based on the availability of messages. Specifically, the processor maintains a queue of received messages, and schedules chares to work based on which chares have messages in the queue. Therefore a chare with no messages will not be scheduled to execute. Being message-driven makes chares an ideal candidate for representing the entities modeled in a PDES. This is because LPs only have work to do when there are events scheduled for them to execute, and LPs without events should not be scheduled to perform any work until an event is scheduled for it. Figure 2.1 (right) shows the process level message queues which contain received messages directed to it's local chares. The runtime system removes messages from these queues, and executes the appropriate method on each message's destination chare. The reason this is effective is due to the previously mentioned notion of over-decomposition. With each process having multiple chares, the runtime has the flexibility to ignore chares without work to do and prioritize CPU time for those chares

that are actively receiving messages. Furthermore, because the communication is one-sided and asynchronous, the runtime system is free to schedule other objects to execute their own work while messages are in flight. This achieves an adaptive overlap of communication and computation based on application characteristics.

Finally, the chares in Charm++ are migratable, i.e., the runtime system can move the chares from one process to another. Moreover, since chares are the only entities visible to the application and the notion of processes is hidden from the application, the migration of chares can be done automatically by the runtime system and transparently to the application developer. This allows the runtime system to enable features such as dynamic load balancing, checkpoint/restart, and fault tolerance with very little effort by the application developer. Complex PDES models can take advantage of these features, thus providing a solution to problems posed by long running dynamic models. In particular, this thesis will focus heavily on the dynamic load balancing aspect of the runtime system in Chapter 5. Charm++ has a robust dynamic load balancing framework, enabled by migratable chares, which allows applications to dynamically balance load across processes during execution using a suite of different load balancing strategies with different characteristics. 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.

2.2 ROSS

ROSS is a massively parallel PDES implementation developed at RPI, with the capability of running both conservative and optimistic synchronization protocols [33, 34, 35]. For optimistic synchronization it utilizes the Time Warp protocol, where the specific mechanism for recovering from causality violations is based on reverse execution in order to avoid the high memory overheads of the state-saving approach [36]. 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 a state-of-the-art simulator 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 [34]. The peak event rate in these runs was in excess of 500 billion events per second. In addition to favorable benchmarking results for the PHOLD synthetic benchmark they have regularly demonstrated scalable performance for a number of different network models [37, 38].

Using ROSS's massively parallel simulation capability, many HPC system models have been developed as part of a DOE co-design project for future exascale systems. These include models for the torus and Dragonfly networks and the DOE CODES storage simulation framework. The torus model has been shown to simulate 1 billion torus nodes at 12.3 billion events per second on an IBM Blue Gene/P system [39]. The Dragonfly model has been scaled to simulate 50 million nodes, with a peak event rate of 1.33 billion events/second using 64K processes on a Blue Gene/Q system [40].

In some of our earlier work, ROSS is reimplemented on top of Charm++ in order to take advantage of the adaptive and asynchronous nature of the Charm++ runtime system [32]. 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 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. The simulator was then further evolved into the Charades simulation engine, which provides a flexible platform for the experiments and research presented in this thesis. This is discussed in detail in Chapter 3.

2.3 MACHINES

All experimental runs for this thesis were done on Vesta and Mira, the IBM BlueGene/Q machines at Argonne National Laboratory, or Blue Waters, the Cray XE6 machine at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. For Vesta and Mira, Charades was built on Charm++'s PAMI communication layer, and all runs used the full 64 threads per node. Each thread had its own Charm++ process. For Blue Waters, Charades was build on Charm++'s GNI communication layer, and all runs used the full 32 threads per node. As with the BG/Q runs, each thread ran its own Charm++ process.

2.4 MODELS

To develop a better understanding of the impact of the techniques explored in this thesis, we focus on variations of three diverse models with varying communication and load char-

acteristics: PHOLD, Dragonfly [40], and Traffic [41]. Here we describe each model and the configurations used throughout the majority of this thesis. Chapter 3 is based on previous work with the ROSS team, and as such uses slightly different model configurations, described in detail in that chapter.

2.4.1 PHOLD

PHOLD is arguably the most commonly used micro-benchmark in the PDES community [34, 42, 43]. 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. 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 most of the experiments in this thesis 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 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.

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.

2.4.2 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 [32] and [40]. 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 thesis we experiment with four different communication patterns: Uniform Random (Dragonfly Uniform) where each message sent goes to a randomly selected MPI process, Worst Case (Dragonfly Worst) where all traffic is directed to the neighboring group in the Dragonfly topology, Transpose (Dragonfly Trans) where message is sent to diagonally opposite MPI process, and Nearest Neighbor (Dragonfly Nbor) where each message is sent to an MPI process on the same router. The particular network configuration used in most of this thesis 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.

2.4.3 Traffic

The Traffic model used in this thesis is an extension of a traffic model available in the production version of ROSS [41], 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.

The first configuration, Traffic Src, represents what traffic may look like after a large 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 second 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 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.

CHAPTER 3: EXTREME SCALE MESSAGE-DRIVEN DESIGN

In this chapter, we explore the impact of the underlying parallel programming system on the productivity in development of a PDES system, as well as the performance of such a system. The hypothesis that we will demonstrate is that an asynchronous message-driven parallel programming framework is a better fit for developing PDES engines than the traditional MPI models used by most distributed PDES engines to date. In particular, we show the outcomes of designing a simulator on top of the Charm++ runtime system, and analyze the resulting productivity and performance benefits. In order to do so, we will compare and contrast the design and performance of ROSS and Charades. Much of this work was originally described in [32].

3.1 COMPONENT DESIGN

In this section we will describe the decomposition of the simulator into its core components and how each component maps to each programming model. We will then discuss each component in detail, and describe its relation with the underlying communication infrastructure provided by each programming model. In doing so, we will show that an asynchronous, one-sided messaging system, such as Charm++, is an effective match for PDES.

3.1.1 Parallel Decomposition

As described in Chapter 1, a typical PDES simulation consists of Logical Processes (LPs), and events. Each event occurs at a specific point in virtual time and is targeted at a specific LP. When executing an event, an LP will likely modify its state and may also create new events scheduled for some virtual time in the future and targeted at any other LP in the system. The simulator must manage all LPs and events, ensuring that events get delivered to their target LPs and that the execution is scheduled in such a way that events are processed in order of increasing virtual time.

Decomposition in ROSS

In ROSS, three main data structures are used to implement this process as shown in Figure 3.1 (left): Processing Elements (`PE`), Kernel Processes (`KP`), and Logical Processes (LP). It is important to point out that each of these three components are passive C struc-

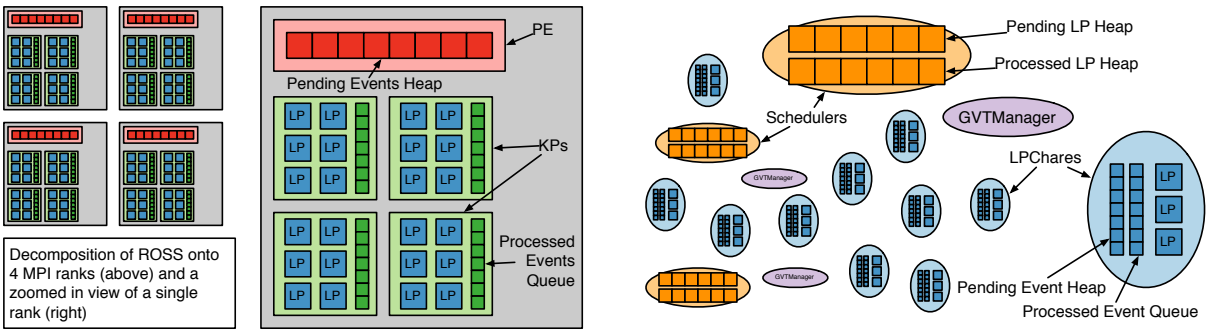

Figure 3.1: User view of decomposition in ROSS (left) and Charades (right). Boxes are regular C structs, and ovals are chares (known to the CHARM++ runtime system). In ROSS, the users view requires explicit knowledge of the mapping of entities to hardware, where in Charades the runtime system manages this mapping.

tures which hold data. The underlying runtime/messaging layer (in this case MPI) has no knowledge of these data structures.

There is a single `PE` object per MPI rank, where each MPI rank is usually associated with a single hardware core or SMT thread. Each `PE` object contains process specific data used when coordinating three key responsibilities of the simulation engine. First, the `PE` contains a single queue of preallocated events used by the simulator on its MPI rank. Second, the `PE` maintains a heap of pending events sent to LPs located on its MPI rank. Third, the `PE` contains data used when coordinating the GVT computation across all MPI ranks.

`KPs` are used primarily to aggregate history storage of multiple LPs into a single object to optimize fossil collection. This history consists of events that have been executed by any LP mapped to the `KP`. Fossil collection refers to freeing up the memory occupied by the events that occurred before the current GVT.

LPs are the key component in actually defining the behavior of a specific model. They contain model-specific state and event handlers as defined by the model writer, in addition to storing some meta-data used by ROSS such as their globally unique ID. The pending events received for LPs are stored with the `PEs` while the events that have been executed by the LPs are stored with the `KPs`.

The organization of ROSS in MPI, as described above, is mainly driven by the process-based model typical of MPI programs. On each MPI process, there is a single flow of control running the ROSS scheduler function which manages the interaction between a single `PE` and a collection of `KPs` and LPs as shown in Figure 3.1 (left). This places the burden of handling communication and scheduling computation for all simulation tasks on the ROSS simulator itself.

Decomposition in Charades

As described earlier, CHARM++ is based off the idea of parallel objects communicating asynchronously, with scheduling dictated by message-driven execution. By utilizing the CHARM++ communication model, we hope to take advantage of the adaptive scheduling of communication and computation to hide fine-grained message latency more effectively than with hand written scheduling code. In order to do so, the CHARM++ runtime system must be made aware of the key data structures that we want it to manage by elevating them to the level of chares. Chares are parallel objects, and first-class citizens in the CHARM++ runtime system. The runtime system is able to manage and schedule execution of each chare, messaging between chares, and the location of each chare. The location management is also key to the second factor we want to design towards: migratability. Since the runtime system manages the location of chares, it is also able to migrate them during execution, meaning we no longer have a static mapping of software components to hardware. When designing the decomposition we want to enable the migration of LPs, as they are the main component of a simulation, while still ensuring that the model can use LPs without needing to have knowledge of the dynamic mapping of LPs to processors. As shown in Figure 3.1 (right), this results in a decomposition with 3 key entities: `LPChares`, `Scheduler` chares, and `GVTManager` chares.

In order to make LPs migratable, `LPChares` now encapsulate everything an LP needs to execute in a single chare. Each chare may encapsulate one or more LPs. The chare contains the LPs' states in addition to storing both the pending and past events for its LPs. The CHARM++ runtime system allows communication of events (and anti-events) directly between `LPChares` as one-sided asynchronous messages. Upon receiving an event, an `LPChare` handles any causality violations or event cancellations immediately by performing the necessary rollbacks, and enqueues the new event into its pending queue. This has the additional effect of localizing the queuing and rollback logic to an LP itself, rather than requiring additional code in the `Scheduler` to deal with event management. The mapping of LPs to `LPChares` is left to the user so that a model-appropriate scheme can be used, as was the case for mapping LPs to `KPs` in ROSS. Note that one important difference here is that `KPs` were not visible to the underlying runtime system, MPI, whereas `LPChares` are visible to CHARM++. This allows the mapping of `LPChares` to hardware resources to be handled by the runtime system. This gives the the runtime system the ability to migrate LPs as needed which enables features such as checkpointing and dynamic load balancing, without requiring the models to change how they address LPs.

`Schedulers` and `GVTManagers` handle the underlying simulator logic responsible for syn-

chronizing the simulation and managing memory via GVT computations. They are implemented as a special collection of chares in CHARM++ called *groups*. Groups are a collections of chares with two distinct properties. First, each group has exactly as many chares as the number of CHARM++ processes that the application is being run on. Second, the chares in each group are mapped to hardware such that there is exactly one chare from each group on each CHARM++ process. `Scheduler` chares coordinate execution of events by `LPChares` on the same process by maintaining a heap of LP tokens. Each LP token has a pointer to an `LPChare`, and the timestamp of its earliest unexecuted event. The heap is sorted by these timestamps in ascending order. This aims to approximate causal ordering by always attempting to execute the earliest known event on each process. The `Scheduler` also manages timing of other parts of the simulation such as the GVT computation, dynamic load balancing, and fossil collection. The `Scheduler` is no longer responsible for handling any of the communication logic, because unlike in ROSS, all communication is handled by the runtime system. Communication progress, and how to schedule communication and computation can now be handled adaptively, freeing up the Charades `Scheduler` to only manage simulator-specific tasks.

The `GVTManager` chares are responsible for actually carrying out the GVT computation when signaled by the `Schedulers` to do so. Once again, messaging between `GVTManager` chares used to compute the GVT is managed by the runtime, and can be overlapped with other work in the `Schedulers`, `LPChares`, or internal runtime system work such as quiescence detection, which is a critical part of the GVT computation and will be discussed in more detail later. The `GVTManager` and `Scheduler` chares interact with each other through a set API, which makes it simple to extend the simulators capabilities by creating different GVT algorithms, as shown in Chapter 4.

Expected Cost of Basic Operations

One of the main changes from the ROSS to Charades is the two-level queue used in Charades to manage execution of pending events. In ROSS, all pending events on a single `PE` are in a single heap and the expected cost to add, remove, or execute an event from the `PE`'s pending heap is $\log N$ when N is the number of events in the `PE`'s pending heap. In the Charades, all event heaps are stored in the `LPChares` and only store events for its own LPs, so the cost of any operation on these heaps is approximately $\log \alpha N$ where $\alpha < 1$ is the proportion of a processors events owned by the `LPChare` in question. In some cases, the addition or removal of an event will change the minimum time of an `LPChares`, requiring an update in the `Scheduler` heaps which incurs an additional $\log C$ operation where C is the

number of chares on the processor. Thus, the expected cost to add, remove, or execute an event in the Charades is at most $\log \alpha N + \log C$, which in cases where N is much greater than C, or when the events are divided roughly evenly among the C chares is approximately $\log N$. This is on par with the cost of event operations under the ROSS decomposition, however in many cases we expect the cost to be even lower, since many operations will not require updating the `Scheduler` heaps and will therefore cost $\log \alpha N$ where $\alpha < 1$.

3.1.2 Scheduling

Now that we have the decompositions described above, we look at the scheduling done by each simulator. In general, each simulator must have some way of scheduling event execution alongside other simulator tasks such as fossil collection, GVT computation, and possibly explicit management of communication resources.

Scheduling in ROSS

In ROSS, the scheduler is simply a function called from `main` after the initialization is completed. There are three types of schedulers supported: sequential, conservative, and optimistic. Algorithm 3.1 provides pseudocode that shows the simplified behavior of the optimistic scheduler, which is almost always the preferred option for ROSS. The scheduler is primarily an infinite loop that polls the network for driving the communication, performs rollbacks if they are needed, executes events on LPs in batches of size b, and performs GVT computation if needed. In the conservative mode, the rollback step is skipped, while in the sequential mode only event execution happens. The salient parts to focus on are on lines 3 and 9. The scheduler is required to explicitly handle communication progress by polling the network on line 3, which pulls events and anti-events off the network. For this part, the ROSS scheduler is only checking for communication involving events and anti-events, and any other communication would be ignored. On line 9 the GVT is computed which requires it's own communication and polling for events. In the basic ROSS implementation, a blocking GVT scheme is used, and is discussed later in this chapter.

Scheduling in Charades

As discussed in the previous section about decomposition, in the message-driven design of Charades, the `Scheduler` is now a runtime system managed object with multiple entry points. It is no longer responsible for explicitly scheduling communication, and is broken up

Algorithm 3.1 ROSS Scheduler Pseudocode

```
1: function RUN()
2:     while 1 do
3:         poll_network()
4:         process_causality()                    ▷ Performs rollbacks and cancellations
5:         for i = 1, ..., b do                   ▷ b is a runtime parameter for batch size
6:             execute_next_event()
7:         end for
8:         if ready_for_gvt then
9:             compute_gvt()                      ▷ Blocks until completion
10:            collect_fossils()
11:        end if
12:    end while
13: end function
```

into pieces which are called by the runtime system based on availability of work and data in the form of messages. The runtime system schedules computation and communication for all chares, including `Schedulers`, as messages become available for them to execute. A single scheduler iteration is represented as a single message to the `Scheduler` chares in Charades, which can be overlapped with work and communication for the `GVTManager` and `LPChares`. The pseudocode is shown in Algorithm 3.2. In this chapter, we focus on a simplified version of the `Scheduler` that works with the same synchronous GVT algorithm we have tested in ROSS. In Chapter 4, we will discuss the extended version of the `Scheduler` that works with more complex GVT algorithms.

As shown in Algorithm 3.2, the Charades `Scheduler` is broken up into two main pieces, each of which is encapsulated in a CHARM++ message that can be scheduled by the runtime system. The first part is the RUN function, which contains a single iteration of event execution (still in terms of a batch of b events at a time). After the execution of b events, the `Scheduler` does one of two things. In most cases, it will not be time to execute a new GVT, so the `Scheduler` will simply move on to the next iteration by sending itself another RUN message (line 8). On this line, `thisProxy` is just the global handle to the calling object (much like `this` is a pointer to the calling object). This message will go into the runtime's message queue along with all other messages for chares on the same process, and the next `Scheduler` iteration will only be run once the message gets to the front of the queue. In between iterations the runtime is free to handle communication and schedule any other tasks it wants, such as messaging between LPs or for the GVT computation. The Charades `Scheduler` itself does not ever explicitly need to progress communication.

If, at the end of an iteration, the `Scheduler` detects it is time to compute a new GVT,

Algorithm 3.2 Charades Scheduler Pseudocode

```
1: function RUN()
2:     for i = 1, ..., b do                          ▷ b is a runtime parameter
3:         execute_next_LP()                         ▷ Based on sorted heap of LPs
4:     end for
5:     if ready_for_gvt then
6:         gvt_manager->begin_gvt()
7:     else
8:         thisProxy.run()                           ▷ Send myself a run message
9:     end if
10: end function
11:
12: function GVTDONE(gvt)                            ▷ Called by GVTManager
13:     collect_fossils()
14:     thisProxy.run()                               ▷ Send myself a run message
15: end function
```

it starts the GVT computation (line 6) instead of scheduling the next iteration. This involves a call to the `GVTManager`, which will then execute whichever GVT algorithm it was configured to perform. Nothing is required from the `Scheduler`, as the entire computation is orchestrated by the collection of `GVTManager` chares, and the communication between them is handled by the runtime along with the communication of any events still in flight. Once the `GVTManager` has computed a new GVT, it calls the GVTDONE method on the `Scheduler`, passing in the new GVT. The `Scheduler` then performs fossil collection and restarts itself with a RUN message (line 14).

The `Scheduler` can also handle scheduling of more advanced features such as load balancing, and checkpointing, but for simplicity we have omitted those for the moment. As mentioned previously, the management of event logic is now contained within the `LPChares`, which further simplifies the `Scheduler` code and again allows for more overlap in execution because the logic for dealing with events is only performed when events are received by each `LPChare`, and only for the particular `LPChare` that receives the event. In fact, the `Scheduler` is entirely isolated from the details of both event sending and the particular GVT computation. This allows for different GVT algorithms to be developed without requiring any change to the `Scheduler` code to support new algorithms or communication patterns, and as we will demonstrate in 4, the GVT algorithm can completely interoperate with the `Scheduler` as well, due to the fact that all of the work chunks are encapsulated in messages scheduled by the runtime system. `Scheduler` subclasses exist for sequential, conservative, and optimistic scheduling protocols, and the correct `Scheduler` is initialized during simulation

startup based on runtime parameters.

3.1.3 Communication Infrastructure

PDES requires communication across processes to deliver events generated by LPs on one process for LPs on different processes. Furthermore, there is also potential communication required for other simulation components such as the GVT computation. In this section we are mostly going to be discussing how the rest of the simulator interacts with and manages communication, however, it is also worth mentioning the actual mechanisms by which communication across the network is handled. Both MPI and Charm++ use low level hardware primitives specific to the machine they are running on under the hood. For example, in results presented later in this chapter, the PAMI network interface is used by both messaging layers for communication across nodes on the IBM Blue Gene/Q machine, Vesta at Argonne National Laboratory. For the ROSS, the simulator itself is in charge of scheduling communication and computation by making calls to MPI, which uses the PAMI interface for actual communication. The Charades simulator passes all scheduling tasks down to the Charm++ runtime system, which then uses PAMI for actual communication. Similarly, in later chapters of this thesis, Charm++ utilizes the GNI network interface for runs done on Blue Waters.

Communication in ROSS

In ROSS, the delivery of events is performed using point-to-point communication routines in MPI. Since MPI's point-to-point communication is two-sided, this requires both the sending and receiving process to make MPI calls in the ROSS simulator. The complexity and inefficiency in this process stems from the fact that the events are generated based on the model instance being simulated, and thus the communication pattern is not known *a priori* by the simulator. Whenever an LP, say *srcLP*, generates an event for another LP, say *destLP*, the process which hosts *destLP* is computed using simple static algebraic calculations and the sender process issues a non-blocking `MPI_Isend`. Since the communication pattern is not known ahead of time, all MPI processes post non-blocking receives at frequent intervals that can match to any incoming message (using `MPI_ANY_TAG` and `MPI_ANY_SOURCE`). Periodically, the scheduler performs network polling to test if new messages have arrive for LPs on the current process. Both of these actions (posting receives and periodic polling) lead to high overheads. In addition to communication for events, ROSS must also explicitly manage communication for the GVT computation. The specific MPI calls made here depend on the

GVT algorithm implemented, and may also require polling for events as well. The GVT computation and required communication is described more in later sections.

Communication in Charades

In Charades, both LPs and their events are encapsulated in `LPChares` which the runtime system knows about. All communication from Charade's perspective occurs between chares, not processes. Using the above example, when *srcLP* generates an event for *destLP*, instead of looking up the process which hosts *destLP*, Charades instead looks up the chare index (using similar algebraic computations). The event message is then handed off to the runtime system which handles the location lookup for the destination chare (which may change dynamically due to migrations from load balancing, for example), and sends the message to the host process. When the message is received by the host process, it is enqueued in the runtime's internal message queue along with any other messages it has received, including `Scheduler` messages as described in the previous section. At some point in the future, the message will be pulled from the queue and executed on the target `LPChare`, which will check for and correct causality violations, and enqueue the event in its pending queue. Moreover, since communication in Charm++'s programming model is all one-sided, there is no need for Charades to post receives on the destination processes. In fact the Charades `Scheduler` is completely unaware of any communication happening between LPs, and all event handling, queuing, and causality checks are completely encapsulated within `LPChares`.

This is precisely what we mean when we say that Charades runs in a message-driven paradigm. It defines a set of message handlers which are only executed as messages that trigger them are pulled from the underlying message queue in the runtime system. The actions of the simulator are entirely driven by what messages are available. This is in contrast to the paradigm of ROSS, in which the simulator is in charge of actively polling the network for more messages at regular intervals, and performs the same tasks in a loop repeatedly, regardless of the messages that are available. It requires the simulator to explicitly manage communication, and as we will show in Section 3.3, message-driven execution provides improved performance by more effectively scheduling computation. This comes largely due to a more effective overlap of communication and computation when using the more dynamic message-driven scheduling when compared to the static scheduling done in ROSS.

Algorithm 3.3 ROSS Blocking GVT

```
1: while sent != recvd do
2:     poll_network()                ▷ Pulls events from network, which updates counts
3:     MPI_Allreduce([sent, recvd], MPI_SUM, MPI_COMM_WORLD)
4: end while
```

3.1.4 GVT Computation

Chapter 4 describes in detail the various GVT algorithms we have explored and developed as part of this research, as well as their implications on simulation characteristics and performance. We acknowledge that there is a wide body of work on other GVT algorithms, but in this chapter our focus is on a direct comparison of the scheduling and messaging paradigms. As such, we only focus on the basic blocking GVT algorithm present in both ROSS and Charades. This section describes at a high-level how the `GVTManager` chares fit into the rest of the Charades design, and how they compare to the original MPI implementation in ROSS.

Computation of the GVT requires finding the minimum active timestamp in the simulation. Hence, while computing the GVT, all of the events and anti-events which have not been executed yet have to be taken into consideration. In algorithms studied in this chapter, this means that any event that has been sent by a process must be received at its destination before the GVT is computed. Without such a guarantee, we are at the risk of not accounting for events that are in transit.

GVT Computation in ROSS

In ROSS, the GVT computation is called directly from the main scheduler function, and invokes a number of blocking MPI collective calls in order to ensure correctness. Pseudocode is shown in Algorithm 3.3. The ROSS simulator maintains two counters: events sent and events received. During the GVT computation, all other execution stops while the simulator makes a series of `MPI_Allreduce` calls interspersed with polling of the network, until the global sum of the two counts match, ensuring that all events that were sent have been received. Then, one last `MPI_Allreduce` is performed to find the minimum timestamp, followed by fossil collection and the resumption of event execution. Like in the ROSS scheduler, we again see that the GVT algorithm must also explicitly manage communication of events by polling the network. In this case, the GVT algorithm just pulls events off the network in order to ensure the minimum is accounted for, but does nothing to handle causality violations caused by any of these new events.

GVT Computation in Charades

Much like the `Scheduler` chares, the `GVTManager` chares in Charm++ have a more narrow focus than the GVT computation in ROSS since they no longer have to manage any communication explicitly. Chapter 4 goes into more detail on the interaction between the `GVTManager` chares and the `Scheduler` chares, as well as the different GVT algorithms employed by the `GVTManagers`. For the results in this chapter we will just describe the basic blocking algorithm, and how its implementation in Charades differs from its implementation in ROSS. In Charades, the `GVTManager` chares on each processor are informed by the `Scheduler` chares on the same process when to begin the GVT computation. At this point, they initiate a process called Quiescence Detection (QD) in Charm++ which uses runtime information to efficiently determine when all sent messages have been received [44]. A more detailed discussion of QD is deferred to Chapter 4 as well. Here, QD serves the same purpose as the repeated `MPI_Allreduce` calls in the ROSS version of the algorithm. While waiting for quiescence, the runtime system continues to receive and handle messages in the same manner as before. Messages go directly to their destination LPs which can immediately handle causality violations. Once quiescence is detected, the `GVTManager` chares check for the minimum virtual time on their own process, then perform a global All-Reduce to find the global minimum. Once this minimum is found, the `GVTManager` informs the `Scheduler` of the new GVT, fossils are collected, and execution continues.

3.2 BENEFITS OF THE NEW DESIGN

This section describes benefits of developing parallel simulation engines, especially those with an optimistic synchronization protocol, using a one-sided message-driven paradigm. In many cases this will be accomplished by comparing Charades to ROSS. However, this section is not meant to discuss in detail the differences between Charm++ and MPI. Rather it is intended to show an effective fit of programming model to the PDES paradigm by comparing Charades against another state-of-the-art implementation. We mostly focus on the inherent benefits that come from the programming model. However, we will also touch upon the new features enabled by the Charm++ runtime system. Some of these features will be discussed in more detail in the later chapters of this thesis. The performance comparison between Charades and ROSS is saved for Section 3.3.

3.2.1 Better Expression of Simulation Components

One advantage of using Charm++ for implementing PDES is that the programming model of Charm++ is a clear natural fit of programming model to problem domain. In the Charm++ programming model, as described in Section 2.1, an application implementation is composed of many objects executing concurrently, driven by asynchronous message communication. When a message is received for an object, the runtime system executes the appropriate method on the object to handle the message. This model is analogous to PDES where several LPs are created to represent simulation entities and event handlers are executed on the LPs when events are available for them. Thus LPs and events map naturally to objects and messages in the Charm++ model. Moreover, similar to PDES models, which can have different types of LPs and events to simulate different types of entities and work, there can be many types of messages and objects in Charm++ to trigger and perform different types of computation. Furthermore, this allows the runtime system to automatically manage communication of events between LPs, as well as any communication required for other simulator components. This further reduces complexity of simulation components. As a result, use of Charm++ provides a natural design and easy implementation, which is in part evidenced by the smaller code base discussed in the next paragraph.

3.2.2 Code Size and Modularity

The programming model and runtime system relieve Charades from managing interprocess communication, network polling, and scheduling computation and communication. Because of this, the size and complexity of the code base has been reduced significantly from its ROSS counterpart. In the initial design of Charades, the *source lines of code* (SLOC) count was reduced to nearly half of its original value; from 7,277 in ROSS to 3,991 in Charades. The communication module of the ROSS code base, which contains a large amount of MPI code for conducting data exchange efficiently, has been completely removed, and other parts of the code base were able to be simplified significantly. Even in the current master version of Charades, where features such as dynamic load balancing have been added, and multiple additional GVT algorithms implemented, the SLOC count is still only 4,740 at the time of this writing. The fact that no part of the simulator has to explicitly manage communication also allows for a modular design, where each component only needs to manage its own state and handle messages as the runtime delivers them. This becomes especially useful when looking at the new GVT framework described in Chapter 4. It provides an easy path to implementing new GVT algorithms without requiring changes to LP or `Scheduler`

logic, while still getting all the benefits of the runtime's adaptive scheduling.

3.2.3 Ease and Freedom of Mapping

Charm++ programmers are encouraged to design their applications in terms of objects and the interactions between them, instead of programming in terms of processes and cores. This relieves the programmers from the burden of mapping their computation tasks to processes and cores, and thus allows them to divide work into units natural to the computation. In PDES, this means that a model writer can focus solely on the behavior of the LPs and map them to objects based on their interactions. As a result, the model writer does not have to worry about clustering the LPs to match the number of processes and cores they may be running on now or in the future. Instead, they can choose the number of objects that is most suitable to their model, and let the runtime assign these objects to processes based on their computation load and communication pattern. A concrete example of this is shown in Section 3.3.2 for the Dragonfly network model. More benefits of separating mapping from correctness are also shown for general cases in [45, 46].

3.2.4 Features Enabled by Charm++

In addition to the benefits described above, use of Charm++ enables many new features that would be difficult or infeasible to implement in large MPI applications such as ROSS. Asynchrony and migratability will be the primary focus of later chapters, however we also mention a few others here. Many of these features can be key in allowing the simulator to adapt to varying workloads and characteristics of different models and configurations.

Asynchronous GVT

As described in Section 3.1.4, the computation of the GVT in Charades is encapsulated within a collection of `GVTManager` objects that interoperate with the `Scheduler` objects by virtue of the runtime systems message-driven scheduling. This provides the perfect platform for implementation of GVTs which can run asynchronously with event execution. The algorithms can be implemented as subclasses of the base `GVTManager`, and when instantiated and started by the `Scheduler`, will automatically overlap with the rest of the work from the `Scheduler` and `LPChares`. Chapter 4 is dedicated to exploring a variety of GVT algorithms implemented within this framework.

Migratability

One of the most important benefits of the object oriented programming model of Charm++ is the ability to migrate objects. The use of `LPChares` to host data belonging to LPs enables migration of LPs during execution of a simulation. In terms of implementation, this has been achieved in Charades by defining a simple serializing-deserializing function for `LPChares` using Charm++'s PUP framework [47]. Migratability of LPs leads to two important features: automatic checkpoint-restart and load balancing. When directed by the Charades `Scheduler` the Charm++ runtime system can migrate LPs between processes to balance load, or migrate to disk to checkpoint simulations.

Load Balancing: In complex, long-running models, load imbalance and excess communication can hinder performance and scalability. Dynamic load balancing algorithms can help address this problem [48, 49] but may add complexity to both the PDES engine, or to specific models. The Charm++ runtime system eases this burden by making migratability a central tenet of its design. Migratability of LPs enables the Charm++ runtime system to periodically redistribute LPs in order to balance load among processes and reduce communication overhead. Chapter 5 is dedicated to further exploring dynamic load balancing within PDES.

Fault Tolerance: In addition to automatic load balancing, the migratability of objects also allows the runtime system to do automatic checkpointing, and provides various mechanisms for fault tolerance [50]. Objects can be periodically migrated to disk to form a checkpoint, which the runtime system can use to restart a computation that has failed. Additionally, objects can be replicated by migrating them to the memory of other nodes so that in the case of a node failure, the simulation can use these replicas to automatically restart the computation from the previous checkpoint [51]. All of these features only require the model writer to inform the runtime system on how to serialize any dynamically allocated state that the model uses.

TRAM

One common characteristic of many PDES simulations is communication of a high volume of fine-grained messages in the form of events. These numerous fine-grained messages can easily saturate the networks, and thus increase the simulation time. To optimize for such scenarios, Charm++ provides the Topological Routing and Aggregation Module that automatically aggregates smaller messages into larger ones [52]. In the past, Acun et al. [47] have already demonstrated the benefits of using TRAM for PDES simulations using a simple

PDES mini-application.

SMP

As future machines move towards nodes with very high core counts, it is apparent that applications will have to take advantage of shared-memory programming to achieve the best performance. This can be difficult in MPI applications because programs are written in a process-centric manner and often require interoperation with another paradigm such as OpenMP. However, because CHARM++ applications are written in terms of objects and their interactions, the runtime system can automatically take advantage of shared memory for communication when possible. Simulators written with this in mind could take advantage of both shared-memory and distributed PDES techniques under one unified programming model.

Introspection

Because the runtime system is in charge of location management and messaging, it has a lot of information it can leverage to automatically improve applications while they are running. This extra information may be able to be used to automatically tune PDES specific parameters such as the GVT frequency, or runtime specific parameters such as how often to load balance a given simulation. Examples of this being done in other applications have already been shown here [53, 54].

3.3 PERFORMANCE ANALYSIS

Scalable performance is a major strength of ROSS that has lead to its widespread use in conducting large scale simulations [55, 40]. One of the primary reasons for the unprecedented event rate obtained by ROSS is its MPI-based communication engine that has been fine-tuned over a decade by the developers of ROSS. Hence, despite the benefits inherent to the design of Charades described in the previous section, it is critical that it also demonstrates good performance. In this section, we study the performance of Charades in comparison to ROSS using two of its most commonly evaluated models - PHOLD and Dragonfly [9, 40]. For these comparisons, we have used the latest version of ROSS as of December, 2015 [41].

3.3.1 PHOLD

PHOLD is one of the most commonly used models to evaluate the scalability of a PDES engine under varying communication load. The PHOLD model is described in detail in Section 2.4.1, however in this chapter we use a slightly different configuration based on collaborative work done with the ROSS team.

Experimental Set up

All the experiments have been performed on Vesta and Mira, IBM Blue Gene/Q systems at Argonne National Laboratory. The node count is varied from 512 to 32,768, where 64 processes are launched on each node to make use of all the hardware threads. For ROSS, each process is an MPI rank, and for Charades each process is a CHARM++ process. On BG/Q, due to disjoint partitioning of jobs and minimal system noise, event rate does not change significantly across multiple trials.

In these weak-scaling experiments, the number of LPs is fixed at 40 LPs per process, each of which receives 16 events at startup. The lookahead (L) and the simulation end time (T) are set to 0.1 and 8,192, respectively. The percentage of remote events, (P), is varied from 10% to 100% to compare Charades and ROSS under different communication loads.

Performance Results

Figure 3.2 compares the performance of PHOLD executed using Charades and ROSS on a wide range of node counts. For both the versions, we observe that the event rate drops significantly as the percentage of remote events increase. This is expected because at a low remote percentage, most of the events are self-events, i.e. they are targeted at the LP which generates them. Hence, the number of messages communicated across the network is low. As the percentage of remote events increases, a higher volume of events are communicated to LPs located on other processes. Thus, the message send requires network communication and the amount of time spent in communication increases, which limits the observed event rate.

At low remote percentages (10-20%), both versions of PHOLD achieve a similar event rate irrespective of the node count being used. However, as the percentage of remote events increase, Charades consistently achieves a higher event rate in comparison to that of ROSS. At 100% remote events, Charades outperforms ROSS by 40%. This shows that when communication is dominant, the runtime controlled message-driven programming paradigm of

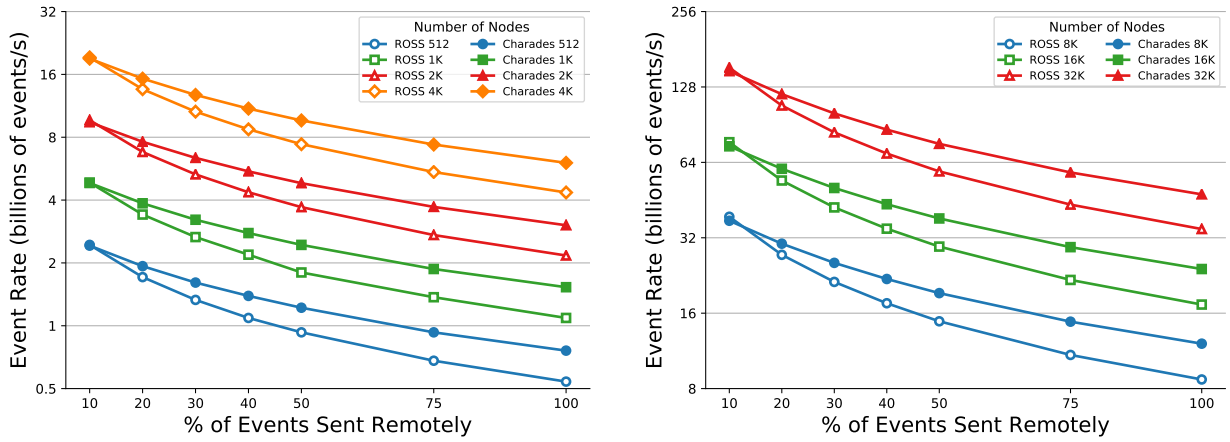

Figure 3.2: Weak scaling experiments for PHOLD with varying amounts of remote communication on up to 32K BG/Q nodes (over 2 millions processes). Charades outperforms ROSS on all node counts by up to 40%

CHARM++ is able to better utilize the given resources. At 50% remote events, which is a more realistic scenario based on the Dragonfly results from the next section, Charades improves the event rate by approximately 28%. Figure 3.2 shows that these improvements in the event rate are observed on all node counts, ranging from 512 nodes (32K processes) to 32K nodes (over two million processes).

Performance Analysis

To identify the reasons for the performance differences presented in the previous section, we used several performance analysis tools to trace the utilization of processes. For Charades we use Projections, a tracing tool built for CHARM++ applications [56]. MPI-Trace library [57] is used to monitor the execution of ROSS. As a representative of other scenarios, we present the analysis for execution of PHOLD on 512 nodes with 50% remote events.

Figure 3.3 (left) shows that for ROSS, as much as 45-50% of the time is spent on computation, while the rest is spent on communication. In this profile, communication is time spent in MPI calls, which include `MPI_Isend`, `MPI_Allreduce`, and `MPI_Iprobe`. A more detailed look at the tracing data collected by MPI-Trace shows that half of the communication time is spent in the `MPI_Allreduce` calls used when computing the GVT, while the remaining half is spent in sending, polling for, and receiving the point-to-point messages. A major fraction of the latter half is spent in polling the network for unexpected messages from unknown sources.

In contrast to ROSS, Figure 3.3 (right) shows that the Charades spends approximately

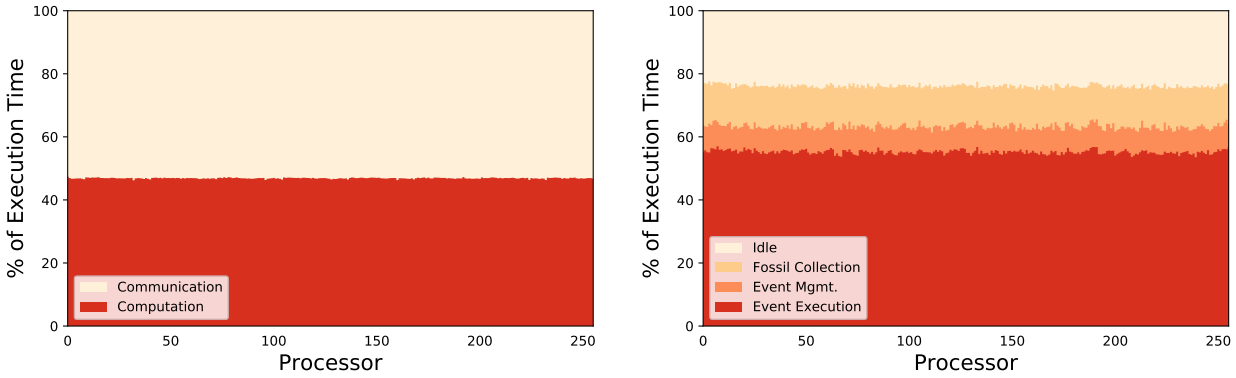

Figure 3.3: Usage profiles for ROSS (left) and Charades (right) executing PHOLD at 50% remote communication on 512 nodes of BG/Q. Communication and runtime system overheads are significantly higher in ROSS (45-50%) compared to Charades (20-25%).

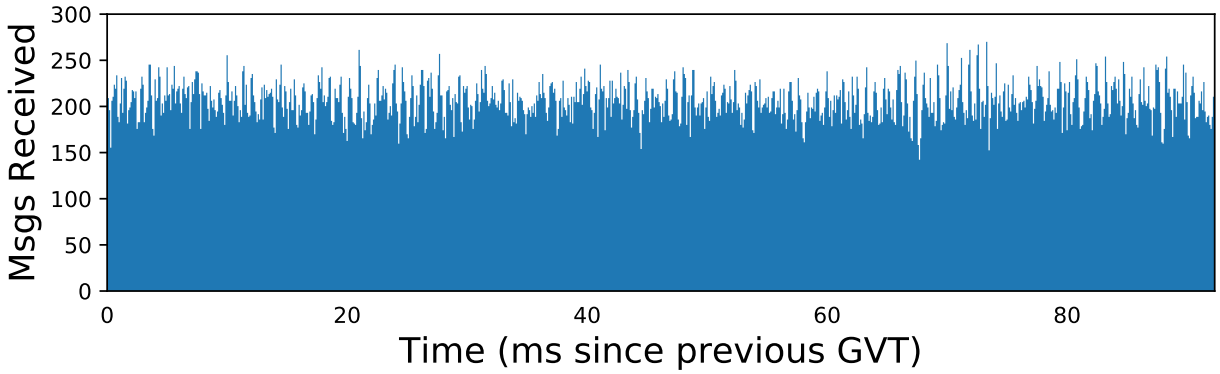

Figure 3.4: The number of messages being processed over the time between two consecutive GVT computations in Charades. Communication is being handled throughout the entire interval by the runtime system.

75% of its time performing computation, while only the remaining 25% is spent in runtime system overhead. In this profile, idle time encompasses any time during which a process is not doing communication. This may be time specifically spent by the runtime system to handle communication and scheduling, or time where there is no work for objects on the process to perform. Since Projections traces are integrated with the Charm++ runtime system, Figure 3.3 (right) provides a more detailed breakdown of time spent doing computation. Approximately 12% of the time is spent doing fossil collection, while 55% of time is spent in the main scheduling loop executing events. The remaining 8% is spent managing events, which entails checking for causality violations and performing necessary rollbacks, as well as pushing received events onto the pending events heap of the receiving LP.

Charades is able to reduce the time spent in communication due to three reasons. First,

Charm++ is a message-driven paradigm, so Charades does not need to actively poll the network for incoming events. This saves a significant fraction of time since Charm++ performs such polling in an efficient manner using lower level constructs. Second, the GVT is computed using a highly optimized quiescence detection mechanism in Charm++, which is based on use of asynchronous operations. Third, since the runtime system schedules execution in Charm++, it is able to overlap communication and computation automatically. Figure 3.4 shows the number of messages received over time during event execution between two consecutive GVT computations in Charades. It shows that throughout execution, messages are actively being handled by the Charm++ runtime system. The runtime system frequently schedules the communication engine, which ensures fast overlapping communication progress.

3.3.2 Dragonfly

The Dragonfly model used in this section is described in Section 2.4.2, and is similar to the one described in [40]. It allows us to study the performance of the two simulation engines on a more realistic and complex application model.

Experimental Set up

Experiments for the Dragonfly model have been done on Mira and Vulcan, an IBM Blue Gene/Q system at Lawrence Livermore National Laboratory. Allocations of sizes 512 nodes to 8,192 nodes have been used, with 64 processes being executed on every node. Just like in the previous experiments, processes are MPI ranks and Charm++ processes for ROSS and Charades respectively. Also, as mentioned above, due to the nature of BG/Q allocations, performance statistics had minimal variability between trials. Two different configurations of the Dragonfly network are simulated on these supercomputers for the Uniform, Transpose, and Nearest Neighbor communication patterns described in Section 2.4.2.

The first configuration of the Dragonfly uses 80 routers per group, with 40 terminals and 40 global connection per router. This results in a system with 256,080 routers and 10,243,200 terminals being simulated, with one MPI process per terminal. The second larger configuration consists of 160 routers per group, with 80 terminals and 80 global connections per router. This system contains 2,048,160 routers and 163,852,800 terminals in total.

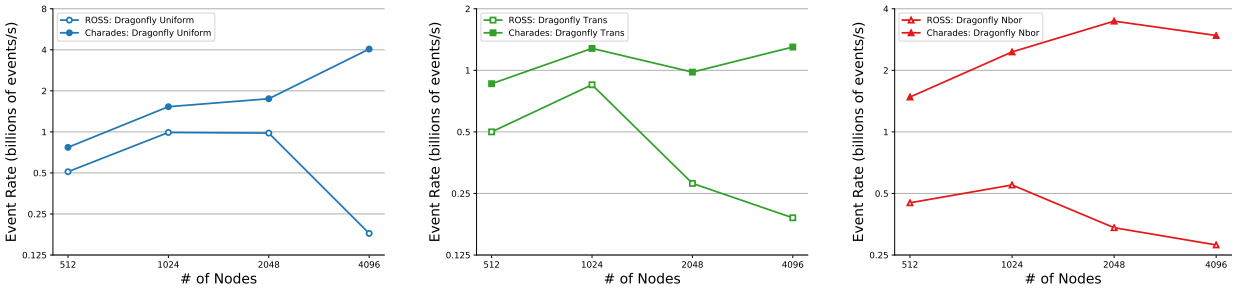

Figure 3.5: Strong scaling of the smaller Dragonfly configuration (256K routers) for each of the three communication patterns.

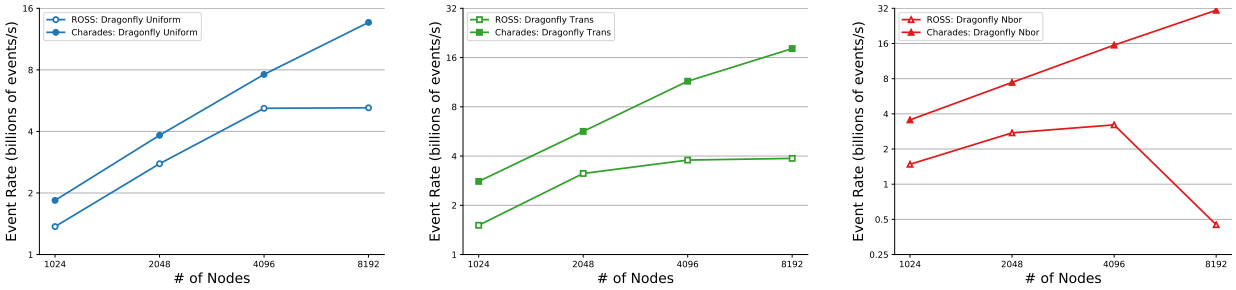

Figure 3.6: Strong scaling of the larger Dragonfly configuration (2K routers) for each of the three communication patterns.

Performance Results

The 256K router configuration of Dragonfly is simulated on 512 to 4,096 nodes. Figure 3.5 shows the observed event rate for these strong scaling experiments using three of the traffic patterns. It can be seen that for each traffic pattern, Charades outperforms ROSS. In each traffic pattern, Charades is able to scale further than ROSS, seeing performance improvements up to higher node counts. In contrast, ROSS sees a dip in performance after 1,024 nodes for all patterns. It is worth noting that, at 2,048 nodes, we are simulating the Dragonfly at near its extreme limits since there are only one to two routers per process. Though the number of terminals is much higher, a large fraction of communication is directed towards the routers in a Dragonfly simulation, which makes it the primary simulation bottleneck.

For the larger 2M router configuration, Figure 3.6 shows distinct patterns in the performance of both simulators. At every data point, Charades achieves significantly higher event rates than ROSS. At 8,192 nodes (half a million processes), Charades has roughly a 4× higher event rate for Dragonfly Trans pattern. For Dragonfly Uniform, 2× improvement in the event rate is observed. The most extreme case is Dragonfly Nbor where we see Charades achieving more than 2× the event rate of ROSS at 1,024 nodes. This advantage increases to 5× at 4,096 nodes, and sky rockets to 60× at 8,192 nodes, mainly because of the drop in

the performance of ROSS.

Performance Analysis

To analyze the performance for the Dragonfly model, we look at two key factors: remote event percentage and event efficiency. In these experiments, both these factors are impacted by how LP mapping is performed in the Dragonfly model. The mapping in both versions of the Dragonfly model is a modified linear mapping that maps each LP type separately. Each execution unit (MPI rank in the ROSS version, or `LPChare` in Charades) is assigned approximately the same number of router LPs, terminal LPs, and MPI LPs. To do so, LPs are assigned IDs to preserve locality. If there are x terminals per router, then the x terminals connected to the first router are given the first x IDs, followed by the terminals connected to the second router, and so on. A similar method is taken for assigning the IDs to MPI LPs.

In an ideal scenario, when the number of router LPs (and hence the number of terminal and MPI LPs) is a multiple of the number of processes, the above mentioned ID assignment guarantees that terminals and MPI ranks that connect to a router are mapped to the same process/execution unit. However, when the number of routers is not a multiple of the number of processes, an even distribution of terminal and MPI LPs to all processes leads to them being on processes different from their router. This results in bad performance for ROSS.

In Charades this issue is easy to handle. Since the number of `LPChares` can be chosen at runtime, we can always ensure that terminal LPs and MPI LPs are co-located with their routers on a single `LPChare`. This advantage of Charades follows from the fact that Charm++ frees the programmer from having to worry about the specifics of the hardware the application is executed on. Instead, the work units of a Charm++ application are the objects, and we have the flexibility to set the number of objects to yield the best performance. In this particular experiment, the best performance was achieved when there was exactly one router (and it's associated terminals and MPI processes) per chare. This change only involved changing the number of objects. The mapping algorithm remains identical to the one used in the ROSS model.

Table 3.1 provides empirical evidence for the issue described above by presenting the average remote communication for each of the traffic patterns simulated in the Dragonfly model. Here, remote communication is the percentage of committed events that had to be sent remotely. It does not include anti-events, or remote events that were rolled back. It is easy to see that Charades requires less remote communication for every pattern, since many of the events are sent among LPs within the same chare. This is especially evident in Dragonfly Nbor where the Charades version has less than 1% remote events in contrast to

	Dragonfly Configuration		
Simulator	Uniform	Trans	Nbor
ROSS	51.40%	34.80%	43.11%
Charades	33.99%	5.97%	0.26%

Table 3.1: Percentage of events which were sent remotely for each Dragonfly traffic pattern and simulator.

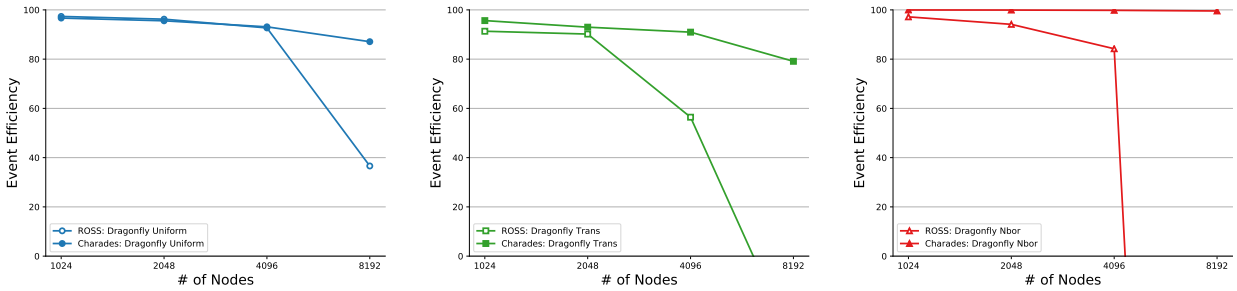

Figure 3.7: Event efficiency for the strong scaling experiments of the larger Dragonfly configuration from Figure 3.6. It uses the event efficiency calculation from ROSS, $1 - \frac{R}{N}$, which allows for negative event efficiencies.

43% remote events for the ROSS version.

The mapping and its impact on remote events also directly affects the event efficiency of each model. Figure 3.7 shows the event efficiency for the different traffic patterns simulated in the Dragonfly model. Here, event efficiency is calculated as $1 - \frac{R}{N}$ where R is the number of events rolled back, and N is the total number of events executed. This differs from how event efficiency is calculated in the rest of the thesis, but is the calculation used within the ROSS simulator. Because of this, we can actually get negative event efficiencies for cases where more events are rolled back than committed. We see that the effect is particularly pronounced for Dragonfly Nbor. In Charades, nearly all events are sent locally, so there is very little chance for causality errors to occur. Because of this Charades maintains 99% event efficiency at all node counts, whereas ROSS drops significantly in event efficiency as the node count increases, eventually dropping as far as -705% at 524K processes. The effects of mapping on event efficiency also provide motivation for the dynamic load balancing work done in Chapter 5.

3.4 CONCLUSION

In this chapter we have shown the suitability of an asynchronous, message-driven programming model for designing and implementing PDES engines. In particular, we compare ROSS, an existing and highly scalable PDES simulator built on MPI, to Charades, a simulator designed on top of the Charm++ runtime system. Charades original design stemmed from ROSS and the two simulators employ similar algorithms and design principles. However, the Charades design in an asynchronous and message-driven programming model exhibits many concrete benefits over the ROSS simulator. The adaptive scheduling of LP execution based on availability of events is able to effectively hide communication overheads by overlapping computation and communication automatically. This yielded better performance and scalability for the models tested in this section. Furthermore, code simplicity and modularity was greatly increased. The size of the code base shrunk dramatically, while still enabling new and useful features for the simulator that we will explore in the later chapters of this thesis. It also allows for flexibility when researching new ideas and algorithms by allowing the runtime system to handle scheduling and communication between multiple distinct modules. This is used to great effect in the next chapter where we are able to design multiple distinct GVT algorithms without changing the core of the PDES scheduler. Each of these GVT algorithms is able to be effectively run in concert with the rest of the simulator, and adaptively overlapped by the runtime due to its asynchronous and message-driven programming model.

CHAPTER 4: GVT COMPUTATIONS

In the previous chapter, we demonstrated the benefits of using message-driven execution to adaptively schedule the communication and computation of a PDES simulation engine. In this chapter, our focus will be on the GVT computation, a key component of any optimistic PDES simulation. In [19], the Global Virtual Time (GVT) at real time t is defined as the minimum of all local clocks at real time t and of the timestamps of all event messages in-transit at real time t. This means that no LP (or processor) will ever receive an event with a timestamp less than the GVT. This is important for two reasons. First, without GVT there is no guarantee that any progress has been made by the simulation. Second, events that have been previously executed with timestamps less than the GVT can have their irreversible effects committed and their memory reclaimed. The computation of the GVT requires global information about the simulation state, and must occur frequently to ensure that enough event memory is reclaimed to prevent simulations from running out of memory. Because of this, it is important for the GVT computation to incur as little overhead as possible, otherwise it can become a significant bottleneck.

In Section 4.1 we describe the basic GVT framework implemented in Charades, and how it relates to the message-driven execution of Charm++ described in the previous chapter. We then look at three different GVT algorithms in the context of this framework, and compare various characteristics and how they affect performance of each model tested. These algorithms all vary in the overhead they incur, communication required, and how they effect LP behavior during the computation. In particular, we will look critically at how each algorithm differs in synchronization overheads, and event efficiency. As a reminder to the reader, event efficiency in this work is defined as $\frac{E_{committed}}{E_{executed}}$. A low event efficiency indicates that a larger percentage of events were incorrectly executed and therefore had to be rolled back.

The first, and simplest, algorithm we look at in Section 4.2 involves completely halting event execution and communication in order to wait for all in transit events to be received. This removes the need to worry about in transit events when computing the minimum virtual time, but requires event execution to stop entirely during the GVT computation. It also has the additional side effect of bounding optimism, the effects of which are studied in [6]. We will refer to this algorithm as the Blocking GVT algorithm and will use it as a baseline when comparing more complex algorithms later in the chapter. Implementations of this algorithm appear in [33, 34, 58, 20]. It is still currently part of the ROSS simulation system as of the time of this writing as well, and was the algorithm used in the experiments

done in Chapter 3. These implementations commonly rely on counting sent and received events, and using repeated reductions to determine when there are no more in transit events. For our implementation, we utilize an adaptive message counting algorithm that works in conjunction with the message-driven infrastructure to efficiently determine when there are no in transit events. We also analyze various configuration tradeoffs and their effects on key performance characteristics. Understanding these tradeoffs will also give a basis on which to evaluate later algorithms. We then evaluate techniques to reduce the synchronization costs by utilizing asynchrony without changing the algorithms core structure in Section 4.2.2.

We then develop two non-blocking algorithms which aim to completely remove the synchronization cost of the GVT computation. We will see in Section 4.2 that the high synchronization cost of the Blocking algorithm adds significant overhead to the simulation. However, completely removing this synchronization has additional side-effects related to the optimistic execution of a simulation that must also be effectively managed. Furthermore, since the following algorithms do not stop the simulation during the GVT computation they can only provide a conservative estimate of the GVT.

In Section 4.3, we develop a continuous GVT algorithm based on the algorithm proposed by Mattern and implemented by various other simulators [19, 33, 58, 20]. The algorithm executes in alternating phases, and events are tagged with the phase during which they were sent in order to correctly account for in transit events. We will refer to this algorithm as the Phase-Based GVT algorithm. In Perumalla's work [58, 20], their implementation uses explicit scheduling of communication and user-written asynchronous MPI reductions in order to achieve an asynchronous implementation of the algorithm. In ROSS [33], the algorithm was modified into one which more closely resembles our Blocking GVT algorithm, and does not allow for LPs to make progress during the computation of the GVT. Within the Charades framework, the messaging and control of this algorithm is overlapped effectively with the rest of the simulation work by virtue of the asynchronous and message-driven execution of Charm++. It shows improvements for a variety of model configurations over the Blocking algorithm by lowering synchronization costs of the GVT computation. In other cases the lack of bounded optimism, which often results in a lower event efficiency, limits the benefits gained from the lower synchronization costs. Furthermore, the Phase-Based algorithm does not utilize information about the virtual timestamps of events being sent and received, and simply uses message counts to detect phase completion. In the face of speculative execution and event rollbacks, this can result in cases where the GVT estimate is much lower than the actual GVT. This can result in higher memory consumption, more GVT computations, and more collective communication than in the Blocking GVT algorithm.

The final algorithm is the Adaptive Bucketed GVT algorithm, presented in Section 4.4.

It is an algorithm we propose which uses principles of the Phase-Based algorithm as well as ideas presented in POSE [59] and SPEEDES [12]. Importantly, it utilizes knowledge of virtual timestamps of the events it is monitoring to drive progress of the GVT computation. It operates by dividing up the virtual time space into buckets, and is able to adaptively select which buckets to include in each GVT computation. This algorithm is able to operate almost completely independently from the event scheduler, and also exposes ways to mitigate the effects of low event efficiency without delaying event execution or the GVT computation itself. Because it is aware of the virtual times of messages it is also able to better decide when GVT computations should occur, resulting in GVT computations which move more uniformly through virtual time and require less collective communication to complete than the Phase-Based algorithm. As a result it also demonstrates the highest event rates for almost every model tested, and the best scaling out of the three algorithms studied.

4.1 CHARADES GVT FRAMEWORK

In Charades the GVT computation is orchestrated by the `GVTManager`, which is implemented as a special type of chare collection in Charm++ called a group. A chare group has exactly one chare mapped to each processor. The event scheduler in Charades is also implemented as a chare group, which means that each processor has exactly one `Scheduler` chare and one `GVTManager` chare. The `Scheduler` and `GVTManager` on the same processor can call methods directly on one another via pointers. Communication across processors is handled by the runtime system, just as for any other chare. The communication and computation of these chares can therefore be overlapped with one another automatically by the runtime system, as well as overlapped with work and communication of `LPChares`. This is critical in developing GVT algorithms which are intended to run concurrently with the simulation scheduler and event execution.

A common API exists for the interaction between `Scheduler` and `GVTManager` so that different implementations of each can be instantiated and used at runtime. The method exposed by the `GVTManager` which can be called by the `Scheduler` is `gvt_begin()`, and tells the `GVTManager` that the `Scheduler` wants a GVT computation to begin. After calling `gvt_begin()`, the `Scheduler` waits until the `GVTManager` invokes the `resume()` method on the `Scheduler`, which lets the `Scheduler` know that it can continue normal event execution. The call to `resume()` is optional, and is only needed for cases where event execution can continue before the GVT computation completes. Once the GVT computation completes, the `gvt_done()` method is invoked on the `Scheduler` informing the `Scheduler` of the new GVT. At this point the `Scheduler` can perform tasks like fossil collection and event execution

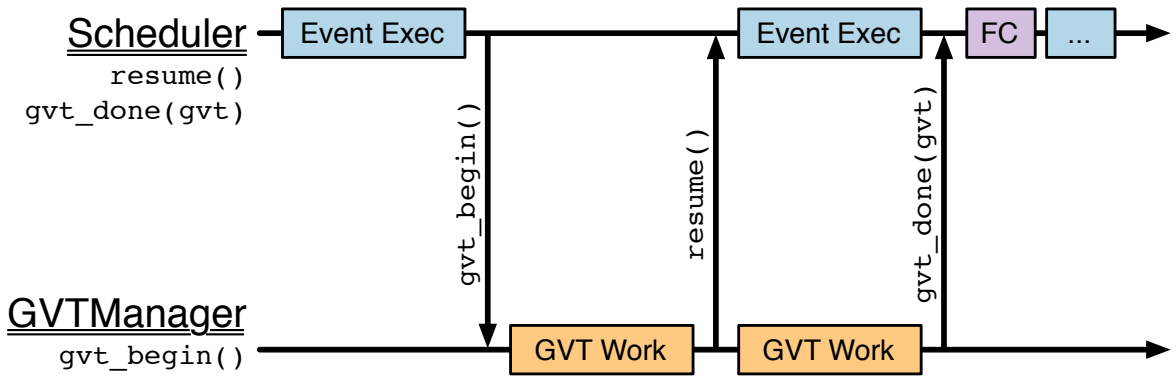

Figure 4.1: Interaction between the `Scheduler` and `GVTManager` chare on each processor. The call to `resume()` is optional and can occur at any point in between `gvt_begin()` and `gvt_done()`.

if it had not been previously resumed. This interaction is diagrammed in Figure 4.1. It is important to note that there is no messaging required between the `Scheduler` and local `GVTManager`, so each of these API functions is called as a normal C++ function. Each of the algorithms discussed in this chapter are orchestrated by a subclass of `GVTManager` which utilizes this API.

In its current form, the `Scheduler` chare on each processor operates completely independently from all other `Scheduler` chares. Similarly, the start of the GVT computation is isolated on each processor and involves no communication with the `Scheduler` chares on other processors. When a `Scheduler` invokes the `gvt_begin()` method, it is simply signaling that it is ready to start the GVT computation. It is up to the `GVTManager` itself to perform any actual coordination with the other processors. Oftentimes this involves a contribution to a reduction, which in Charm++ is one-sided and asynchronous, ie non-blocking. The reduction will complete once each `GVTManager` has contributed, letting the `GVTManager` chares know that every processor is ready to compute the GVT. The call to `resume()` is optional, and can be called at any point after `gvt_begin()` and before `gvt_done()`. For non-blocking algorithms, the `GVTManager` will immediately invoke `resume()` as part of the call to `gvt_begin()`, which allows the `Scheduler` to resume execution for the entirety of the GVT computation. We will also look at a modification to the Blocking GVT algorithm in Section 4.2.2, which resumes the `Scheduler` before the GVT algorithm is completed, but only after it is safe to do so.

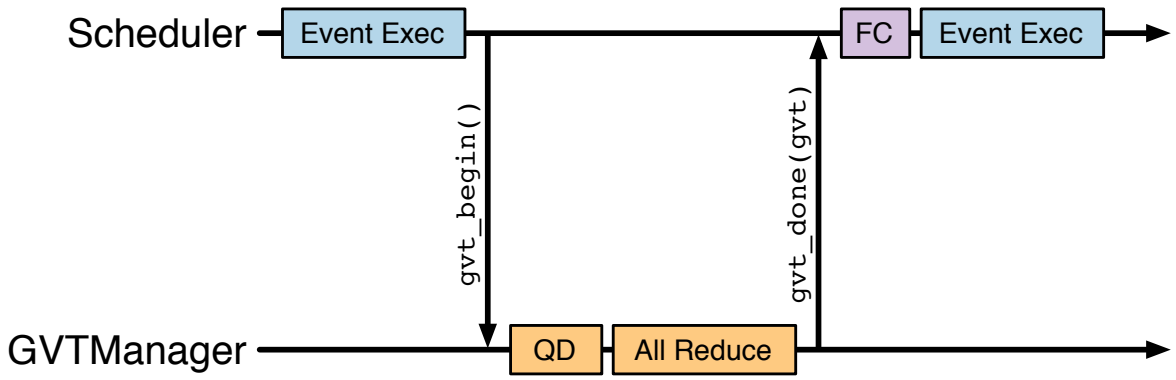

Figure 4.2: Control flow diagram for the basic Blocking GVT algorithm. Event execution is blocked for the entirety of the GVT computation.

4.2 BLOCKING GVT ALGORITHM

The first GVT algorithm we will describe is the Blocking GVT algorithm ported over from the original ROSS code base. The crux of the algorithm is to block event execution until all events have been received on their destination processors. This ensures that all events that have been sent are accounted for and an accurate global minimum timestamp can be computed via a reduction. An implementation of this algorithm is described by Perumalla in [58, 20]. A similar algorithm has also been described and employed by ROSS at large scales [60, 33, 35], achieving good scalability and raw performance when compared to other contemporary simulators. In order to determine when all events have been received both of these implementations rely on repeatedly calling `MPI_Allreduce` to sum up global counts of events sent and received until they counts are equal. Because event execution blocks entirely while the GVT is being computed, this GVT algorithm also has similarities to Bounded Time Warp [6] in that it prevents individual LPs from executing too far in the future. The optimism may be limited to virtual time windows, or like in ROSS, limited by the number of events executed in each window. The high-level flow of the algorithm is diagrammed in Figure 4.2.

4.2.1 Implementation

The Charades implementation of this algorithm primarily relies on Quiescence Detection in Charm++ [44]. Charm++ Quiescence Detection (QD) utilizes an efficient and adaptive algorithm to determine when all messages (in our case this means events) have been received and processed by their destination processors. The algorithm creates a spanning tree of

Algorithm 4.1 Blocking GVT

```
1: function GVTBegin()
2:     if my_processor == 0 then
3:         CkStartQD(QDComplete)        ▷ Start QD, call QDComplete when it finishes
4:     end if
5: end function
6:
7: function QDComplete()                ▷ Called on all procs when QD triggers
8:     lvt = scheduler->min_time()
9:     min_reduction(lvt)
10: end function
11:
12: function MinComplete(new_gvt)       ▷ Called on all procs when reduction completes
13:     current_gvt = new_gvt
14:     scheduler->gvt_done(current_gvt)
15: end function
```

processors, with each maintaining counts of the number of messages it has sent and received. As the lower levels of the tree become idle, they propagate these counts towards the root. Because counts are only propagated when a processor is idle, the algorithm is able to adapt to the load of the system in order to send a small number of control messages and to avoid interfering with application progress. If, at the root of the spanning tree, the counts remain unchanged and equal for two consecutive iterations of the algorithm it can be guaranteed that the system is quiescent (all sent messages have been received). In Charades, this means that all events have been received by their target LPs, all causality violations have been resolved, and the events are stored in Charades specific data structures for later execution as managed by the `Scheduler`. The message-driven nature of the Charm++ runtime system means that the QD library can run efficiently in the background, with its communication automatically overlapped with the communication of events in the simulator.

The full GVT algorithm is shown in Algorithm 4.1. When GVTBegin is called on the `GVTManager` chare on processor 0, it starts the QD library and sets the callback to one that will be sent to all `GVTManagers` once total quiescence is reached (line 3). The GVTBegin call on all other processors is essentially a no-op. Once QD has completed, each `GVTManager` will receive a callback to call QDComplete which queries the local `Scheduler` for the local virtual time (LVT) of the processor (line 8). Then this LVT is contributed to an All-Reduce to find the GVT, which is the minimum of all LVT values across all processors (line 9). Once the All-Reduce completes, the new GVT is known on every process and `gvt_done()` is called on the `Scheduler` chare on every process (lines 13 and 14). This triggers fossil collection

and resumes normal event execution.

This algorithm will serve as a baseline with which to compare the algorithms described in Sections 4.3 and 4.4. Furthermore, careful study of this algorithm reveals a few key improvements that can be made while preserving the basic structure. These changes not only improve performance, but will provide key insights that will aid in more effective orchestration of the GVT as a whole. The first factor we will explore in detail is the trigger, which determines when to start the GVT computation, as well as the frequency of GVT computations.

4.2.2 Performance

The performance and scaling of the Blocking GVT algorithm will be used throughout the rest of this chapter as a baseline against which to compare more complex GVT algorithms. Our focus will be on GVT frequency, its effects on synchronization costs and event efficiency, and scalability.

GVT Trigger Comparison

Because this algorithm blocks event execution entirely, the frequency of GVT computations has a large impact on overall performance. Related to frequency, the metric for determining exactly when to trigger the GVT has significant performance implications as well. Here, we will investigate two different GVT triggers and show how they affect various simulation characteristics. In ROSS, the trigger that determines when to compute a GVT is event count. Specifically, the GVT computation will be triggered on a processor after it has executed X events, where X is a runtime parameter specified by the user. This trigger aims to keep computation time across processors balanced for simulations where events take approximately the same amount of time to execute. The second trigger is based on the idea of virtual time windows present in both YAWNS (for conservative simulations) and Bounded Time Warp (for optimistic synchronizations) [6]. Events are executed in virtual time windows based on a leash from the previously computed GVT. The GVT computation will be triggered on a processor once its LVT is at least T units of virtual time from the previously computed GVT, where T is again a user specified runtime parameter. In both cases, decreasing the size of the interval between GVT computations causes more frequent GVT computations. As we'll show in the following paragraphs, it also has important effects on both the event efficiency of the simulation and the synchronization costs of the GVT computation. With both triggers, we will refer to the amount of time between consecutive

	PHOLD Configuration			
GVT Interval (# of events)	Base	Work	Event	Combo
1,024	98%	97%	54%	54%
2,048	97%	94%	54%	54%
4,096	97%	75%	54%	53%
8,192	96%	63%	52%	46%
16,384	95%	57%	50%	37%

Table 4.1: Event efficiency for various GVT intervals (in number of events) for the count-based GVT trigger.

GVT computations as the GVT interval. The interval is measured in events for the count trigger, and in virtual time for the leash trigger.

Table 4.1 shows how the GVT interval affects the event efficiency of each PHOLD configuration when using the count-based GVT trigger. In the leftmost column, we list the number of events between each GVT computation, and as the number increases we see the event efficiency of each model decrease. This decrease in event efficiency is due to the fact that processors are able to advance further in virtual time. The further a processor gets from the GVT, the more likely it is that an event can arrive which would induce a rollback. However, we see that the sensitivity to GVT interval varies widely depending on the characteristics of the model configuration. For PHOLD Base, which is naturally uniform and balanced, we see that event efficiency is high and the GVT frequency has very little effect on it. For PHOLD Work, which has an imbalance in the time taken to execute each event, we see event efficiency starts high but gets significantly lower as we increase the number of events computed between each GVT. The distribution of events to LPs is the same as in PHOLD Base, so when the interval is small the event efficiency still remains high. For such small intervals, the fact that some events are heavy does not have much impact on event efficiency because the frequent GVTs cause computation to block often. This allows processors with the heavy events to catch up to processors with light events, preventing those processors from getting too far ahead. Even in cases where a rollback is triggered due to a slow processor sending an event to a fast processor, the number of events that need to be rollback is limited by the frequent blocking for the GVT. As the interval increases, processors with lighter-weight events are able to execute much further into the future, which causes a larger number of events to be rolled back when an event from a slow processor arrives. For PHOLD Event, we see the opposite behavior. Because the distribution of events is not balanced across LPs,

	PHOLD Configuration			
GVT Interval (virtual time)	Base	Work	Event	Combo
1	99%	99%	99%	98%
2	99%	94%	96%	92%
4	98%	76%	84%	61%
8	97%	60%	69%	38%
16	96%	55%	64%	34%

Table 4.2: Event efficiency for various GVT intervals (in units of virtual time) for the leash-based GVT trigger. Event efficiencies are generally higher than for the count-based GVT trigger shown in Table 4.1.

executing the same number of events on two different processors no longer means that those processors will progress through virtual time at the same rate. This means processors with less events to execute will consistently get further ahead of processors with more events, regardless of the fact that the GVT causes them to occasionally block event execution. This results in consistently lower event efficiency for PHOLD Event. However, since events all take the same amount of time to execute, the event efficiency is not as affected by GVT interval. For PHOLD Combo, which has an uneven distribution of events and a mix of heavy and light events, we see the worst aspects of both unbalanced configurations. Like PHOLD Event, the event efficiency is low across the board, and like PHOLD Work, event efficiency decreases significantly as GVT interval increases.

Table 4.2 shows how event efficiency changes with GVT interval using the virtual time leash as the GVT trigger. The time intervals for the leash-based trigger in this experiment were chosen to result in a similar number of GVT computations as the count-based trigger. Specifically, the smallest intervals of 1,024 events and 1 unit of virtual time both resulted in approximately 1,024 GVT computations for PHOLD Base. The largest intervals of 16,384 events and 16 units of virtual time resulted in approximately 64 GVT computations for PHOLD Base. When comparing to count-based trigger results in Table 4.1, we see that the performance of the leash-based trigger is similar for both PHOLD Base and PHOLD Work. Both configurations have very high event efficiency when the interval is small. The event efficiency of PHOLD Work decreases as significantly as the interval increases, which again is due to the fact that the amount of work per GVT interval is unbalanced. However, the leash trigger achieves a much higher event efficiency for cases where there is in imbalance in event distribution (PHOLD Event and PHOLD Combo). This is due to the fact that the leash trigger more effectively limits how far ahead any individual processor can get in virtual time.

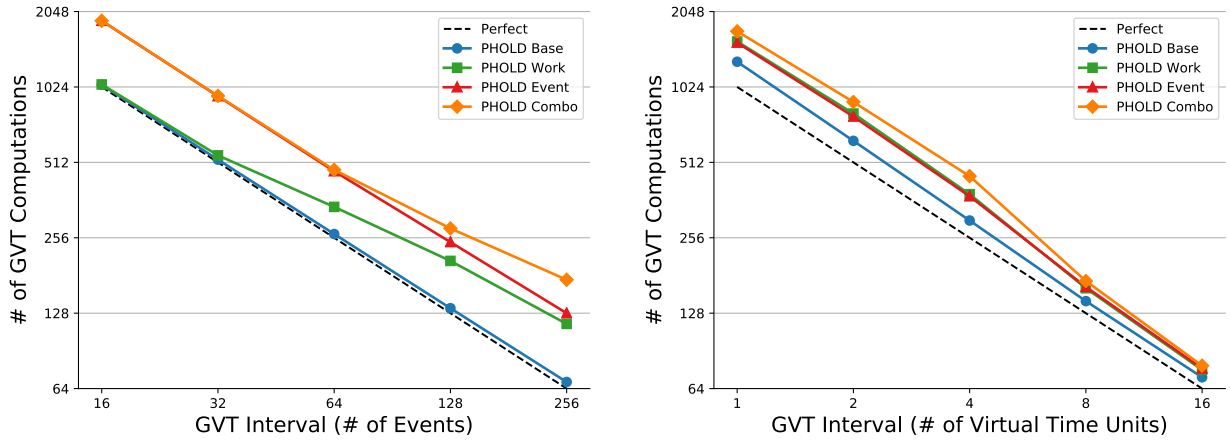

Figure 4.3: The number of GVT computations at varying GVT intervals for both the count-based GVT trigger (left) and leash-based GVT trigger (right).

For example, at the smallest possible interval in this table, any given processor can get at most 1 unit of virtual time ahead of any other. In the count-based trigger, a processor could get arbitrarily far ahead which resulted in many more events getting rolled back. Even in PHOLD Base and PHOLD Work, at small intervals there is a slight improvement in event efficiency over the count-based metric.

In addition to event efficiency, the frequency of the GVT computation also has significant impact on the synchronization costs imposed by the GVT on the rest of the simulation. First, and most straightforward, increasing the interval between each GVT computation results in fewer total GVT computations for a given simulation. Figure 4.3 shows how the number of GVT computations changes as we increase the interval between GVT computations for both triggers. For the most part, the trend is exactly as expected: doubling the GVT interval results in approximately half the GVT computations. However, for the count-based trigger we see that this is not exactly the case for PHOLD Work and PHOLD Combo. This is due to two factors. First, with the count-based trigger, the number of GVT computations is a function of the total number of events executed and the size of the GVT interval. Second, the fact that event efficiency decreases as the interval decreases for both PHOLD Work and PHOLD Combo means that those two models ultimately execute more events as the interval increases. Since the GVT is computed after a fixed number of events, and the number of events executed for PHOLD Work and PHOLD Combo increases at higher GVT intervals, we see a higher number of GVT computations than expected. This is also shown by PHOLD Event, which has a low event efficiency at all GVT intervals for the count-based leash. It requires many more GVT computations that PHOLD Base, which maintains high event efficiency at all GVT intervals. For the leash-based trigger, the number of GVT

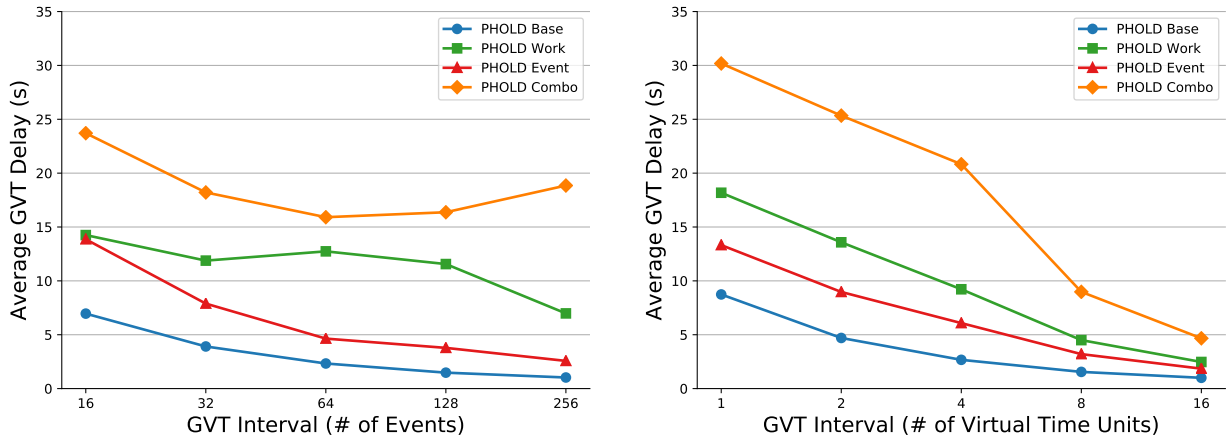

Figure 4.4: The amount of time spent blocking execution on the GVT computation at varying GVT intervals for both the count-based GVT trigger (left) and leash-based GVT trigger (right). This is calculated as the total amount of time the `Scheduler` waits on the GVT computation, averaged across all processors.

computations is not affected by event count, and is solely reliant on the total virtual time executed and the length of the leash. Because of this, the number of GVT computations are much more closely clustered and have a more consistent trend. Even as event efficiency decreases, the number of GVT computations still decrease at an rate inversely proportional with the increase of the GVT interval.

In Figures 4.4 and 4.5, we see how the GVT frequency affects the amount of time the simulator spends blocking event execution. Figure 4.4 shows the total amount of time spent blocking averaged across processors. This is computed by measuring the total amount of time a processor waits on the GVT computation, averaged across all processors. Especially in unbalanced configurations, some processors may trigger a GVT computation significantly before others. These processors will still block event execution until the GVT computation completes. Because of this, the GVT delay encompasses time taken by the GVT computation itself, as well as the amount of time a processor waits for other processors to get to the GVT computation. Figure 4.5 shows the average delay per GVT computation, and it is computed by taking the total delay from Figure 4.5 and dividing by the total number of GVT computations from Figure 4.3.

Overall, we see similar trends as to what was shown by looking at event efficiency and number of GVT computations. The count-based trigger performs fine when the event efficiency is unaffected by GVT interval. Figure 4.4 (left) shows that the amount of GVT delay decreases as GVTs become less frequent for both PHOLD Base and PHOLD Event. For PHOLD Work and PHOLD Combo, the trend is much less favorable and can be attributed

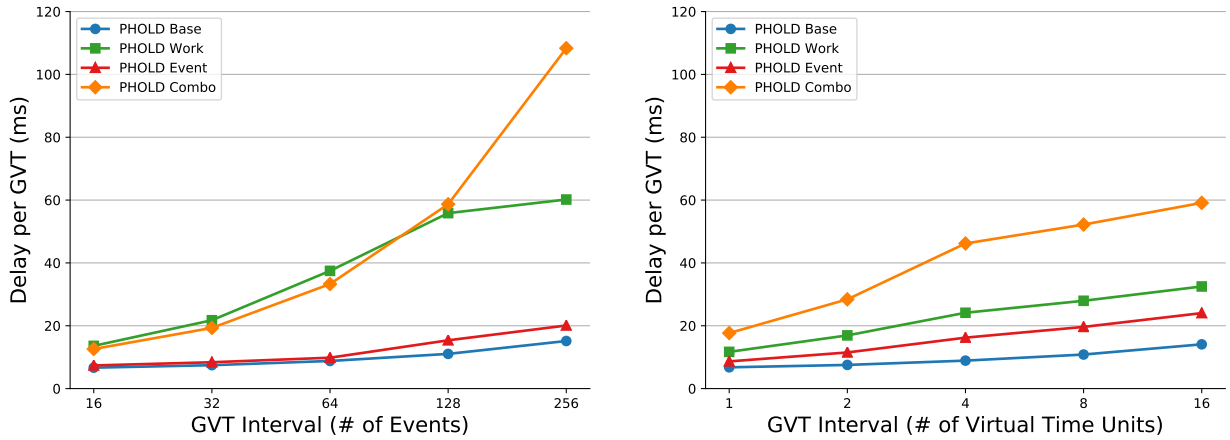

Figure 4.5: The amount of time spent blocking execution for an individual GVT computation at varying GVT intervals for both the count-based GVT trigger (left) and leash-based GVT trigger (right). This is computed as the average total delay shown in Figure 4.4 divided by the total number of GVT computations shown in Figure 4.3.

to a combination of event efficiency and number of GVT computations. As the interval increases, event efficiency significantly decreases. This can result in more work for certain processors, due to the need to rollback and re-execute more events. These processors therefore increase the amount of time other processors spend waiting for the GVT computation to complete. This is exacerbated by the fact that for these two configurations, the number of GVT computations also do not decrease at an inversely proportional rate to the GVT interval. Figure 4.5 also shows that as the interval increases, so does the delay per GVT computation. The increase in interval widens the gap between processors that get to the next GVT computation quickly, resulting in more time blocking on average.

For the leash-based trigger, Figure 4.4 (right) shows much more favorable trends for the GVT delay. For each model, the delay decreases monotonically as the GVTs become less frequent. This is due to the fact that the leash trigger maintains a more consistent decrease in number of GVT computations, and a higher event efficiency. At higher GVT intervals, the leash trigger has less GVT delay for every model in comparison to the count trigger. Figure 4.5 (right) shows that the delay per GVT computation still increases as the GVT interval increases. However, it increases at a less significant rate than for the count trigger.

Figure 4.6 shows the overall effects of all of these factors on event rate for each trigger. We see that for the PHOLD Base model, which is perfectly balanced and uniform, decreasing GVT frequency lowers synchronization cost without a significant hit in event efficiency and therefore overall event rate increases all the way up to the largest intervals tested. For all other configurations however, based on tradeoffs of the aforementioned factors, the optimum

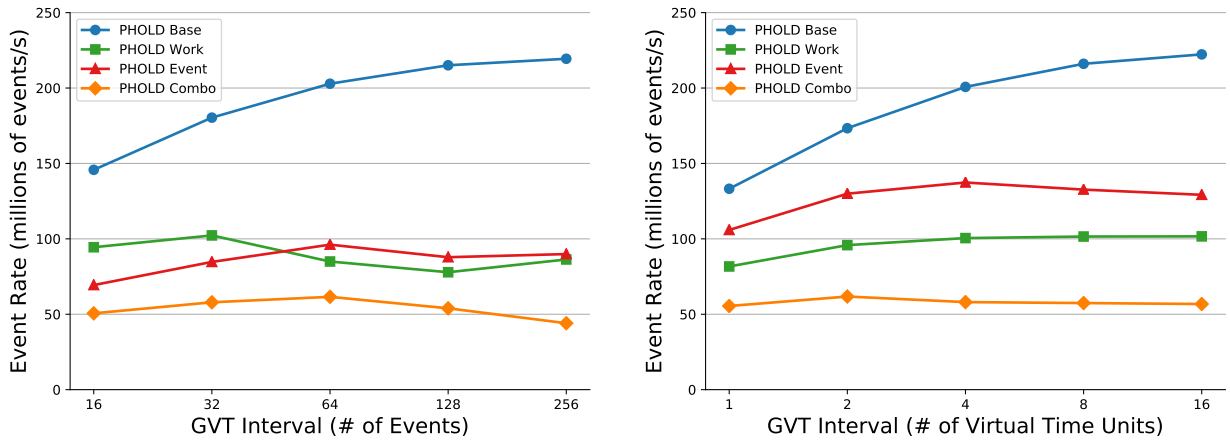

Figure 4.6: Event rates at varying GVT intervals for both the count-based GVT trigger (left) and leash-based GVT trigger (right).

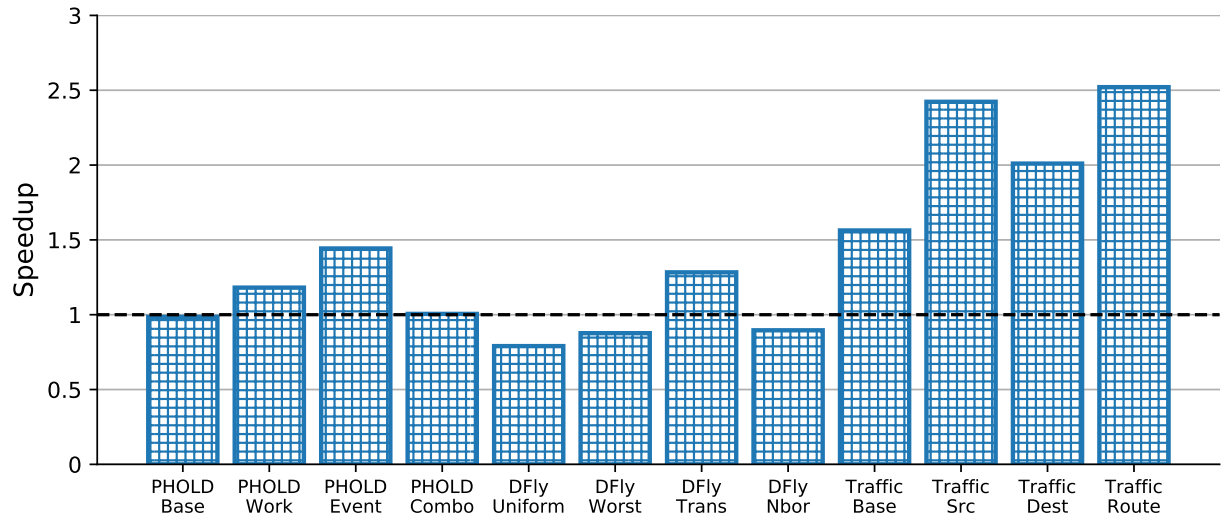

Figure 4.7: Speedup of the Blocking GVT algorithm when using the leash-based trigger compared to the count-based trigger.

	PHOLD				Dragonfly				Traffic			
GVT Trigger	Base	Work	Event	Combo	Uniform	Worst	Trans	Nbor	Base	Src	Dest	Route
Count	97%	75%	54%	54%	74%	39%	31%	68%	46%	12%	51%	11%
Leash	98%	76%	84%	93%	62%	91%	85%	93%	96%	97%	96%	97%

Table 4.3: Event efficiency for the best configurations of each GVT trigger.

	PHOLD				Dragonfly				Traffic			
GVT Trigger	Base	Work	Event	Combo	Uniform	Worst	Trans	Nbor	Base	Src	Dest	Route
GVT Delay (s)												
Count	2.37s	12.69s	4.66s	15.90s	2.66s	3.51s	6.04s	0.26s	5.39s	29.22s	15.02s	39.60s
Leash	2.75s	9.23s	5.96s	25.12s	3.42s	4.73s	7.51s	0.59s	5.27s	22.20s	8.30s	26.35s
% of Execution												
Count	22%	50%	21%	46%	51%	80%	52%	50%	46%	50%	72%	55%
Leash	25%	43%	39%	73%	52%	94%	83%	64%	72%	92%	80%	93%

Table 4.4: Synchronization costs for the best configurations of each GVT trigger, both in absolute time and percentage of total execution time.

performance comes somewhere in the middle. Figure 4.7 shows the overall effect of the leash trigger on event rate by plotting speedup of the optimum configuration for the leash trigger compared to the optimum configuration of the count trigger for each models and configuration. Unlike in the previous figures, where intervals were chosen such that the number of GVT computations was consistent for each trigger, here each trigger and model were tuned independently for best performance. Tables 4.3 and 4.4 show the event efficiency and synchronization cost for each models optimal configuration.

We see in Figure 4.7 that the models which demonstrate the largest speedups generally correspond to the models which see the most significant increases in event efficiency in Table 4.3. The table also shows that in many cases, the optimum leash trigger configuration results in an increase in time spent waiting on the GVT computation as well as a more significant portion of total execution being taken up by the GVT computation. The major out-lier is the Traffic model, where the leash results in both higher event efficiency and lower GVT delay. In this model, the event efficiency is so low on certain processors, that even under the count trigger the extra work for rollbacks causes a similar imbalance across processors that increases GVT Delay for those processors that get to the GVT quickly. However, we also see that the amount of time spent blocking on the GVT still takes up a much greater percentage of overall runtime for the Traffic model. This again demonstrates a tradeoff with the GVT configuration, where using a slightly more expensive configuration in terms of GVT synchronization cost can result in the rest of the simulation taking significantly less time which more than makes up for the added overhead.

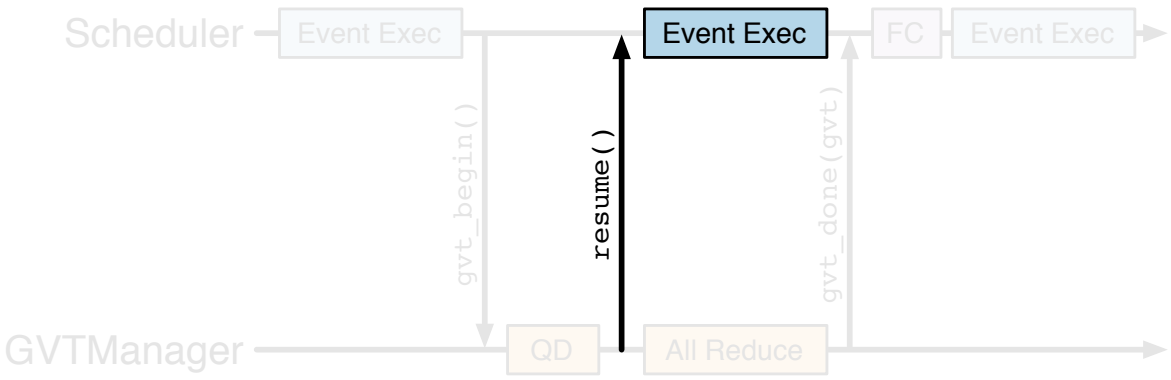

Figure 4.8: Control flow diagram for the addition of the asynchronous reduction to the Blocking GVT algorithm. Event execution is blocked for the QD portion of the GVT computation, but is allowed to continue once QD completes.

Algorithm 4.2 Blocking GVT algorithm asynchronous reduction modification

```
function QDComplete()
    lvt = scheduler->min_time()
    scheduler->resume()
    min_reduction(lvt)
end function
```

Asynchronous Reduction

In analyzing the different triggers, we see a clear relation between synchronization cost and event efficiency. By choosing a more effective trigger, we are able to achieve a more favorable tradeoff. In order to decrease the GVT synchronization cost independently of the GVT frequency, we can exploit asynchronous reductions for the All-Reduce component of the GVT. Observe that once QD completes, each processor knows its local virtual time and can immediately contribute it to an asynchronous reduction (Algorithm 4.1 lines 8 and 9). At this point, if event execution were to resume, it would not effect the result of the reduction. By resuming event execution early, some of the synchronization cost of the GVT computation can be avoided by allowing event execution to overlap with the reduction. Algorithm 4.2 shows a simple modification to the QDComplete function of the Blocking GVT algorithm, which resumes the `Scheduler` as soon as quiescence is reached. This change is also diagrammed in 4.8, with the changes shown darker than the unmodified portion of the algorithm. By looking at the diagram, it becomes clear that the effectiveness of this modification will be largely dependent on the ratio of time spent in QD compared to the reduction.

	PHOLD				Dragonfly				Traffic			
	Base	Work	Event	Combo	Uniform	Worst	Trans	Nbor	Base	Src	Dest	Route
Baseline	2.75s	9.23s	5.96s	25.14s	3.42s	4.73s	7.51s	0.59s	5.27s	22.20s	8.30s	26.35s
w/ Async Redn	2.32s	8.73s	5.50s	24.56s	3.79s	3.07s	5.51s	0.28s	4.16s	20.80s	7.15s	24.66s
% Change	-16%	-5%	-8%	2%	11%	-35%	-27%	-53%	-21%	-6%	-14%	-6%

Table 4.5: The average amount of time that event execution is blocked by the Blocking GVT algorithm with and without the asynchronous reduction modification, as well as the percent change.

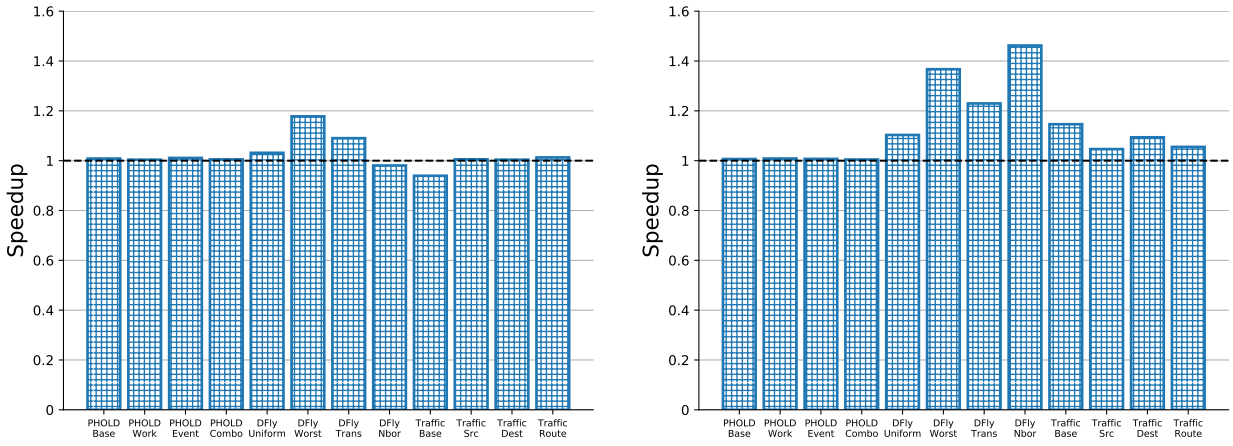

Figure 4.9: Speedup when using the asynchronous reduction for both the count-based GVT trigger (left) and leash-based GVT trigger (right).

To prove that this modification does not effect the correctness of the computation, realize that at any point in the simulation there is some event, e, which has the smallest timestamp, t, out of all events currently in the simulation. Because the algorithm still waits for complete quiescence before starting the reduction, this event e will be on some processor p when it is time to contribute to the reduction. This processor will therefore contribute t to the reduction, since t is the smallest timestamp in the entire simulation. After contributing, p will be allowed to continue event execution which may result in new events being created and sent to other processors. However, the timestamps of these new events will all be greater than or equal to t. These events may arrive on other processors before those processors contribute to the reduction, after they contribute but before the reduction completes, or after the reduction completes. These events may also cause rollbacks when arriving on their destination processors. However, in all cases the result of the reduction, and therefore the new GVT, will still be t. Furthermore, no sent event can result in a rollback to a time before t. Therefore t will be a valid estimate of the GVT.

To evaluate the effectiveness of this modification, we first look at how it affects the time

	PHOLD				Dragonfly				Traffic			
GVT Trigger	Base	Work	Event	Combo	Uniform	Worst	Trans	Nbor	Base	Src	Dest	Route
Count	8.10	6.29	4.54	4.48	0.75	0.28	0.32	12.88	3.49	0.86	2.62	0.79
Leash	7.08	5.60	5.69	2.40	0.79	0.12	0.17	1.40	1.57	0.72	1.64	0.76

Table 4.6: The number of events committed per GVT on average for each simulation configuration and GVT trigger. Models with a higher number of events per GVT tend to see more benefit from increasing event efficiency, where the models with fewer tend to get more benefit from reducing synchronization cost of the GVT algorithm.

spent blocking event execution during the GVT computation. Table 4.5 shows the time spent blocking both with and without the asynchronous reduction modification. It also shows the percent difference in the amount of time blocking. It shows that, for the majority of models, the amount of time blocking is decreased by up to 20%. However, this also reveals that 80% or more of the time spent blocking is during the QD portion of this algorithm. Figure 4.9 shows the effect of the modification on event rate by plotting speedup of each model when using the asynchronous reduction compared to without. We see modest speedups in most configurations, roughly proportional to the decrease in time spent blocking. To understand why certain models, such as Dragonfly and Traffic, show better improvements using the asynchronous reduction we need to look at the fraction of time that the simulation spends actually performing the GVT computation. Table 4.4 shows that Dragonfly and Traffic both spend a larger fraction of their total execution time performing the GVT computation, which means reductions in GVT compute time have a larger impact on overall performance. The table also shows that the leash trigger generally spends a larger fraction of time in the GVT computation than the count trigger. This explains the increased effectiveness of the asynchronous reduction when using the leash trigger. Due to the fairly uniform execution of the PHOLD model, the optimum configurations generally perform fewer GVT computations while still getting good performance. The more complex execution of Traffic and Dragonfly require more GVT computations, which results in the higher fraction of execution time being dedicated to the GVT computation. Table 4.6 further demonstrates this by showing the number of committed events per GVT computation for each model. These results reinforce the observation that more complex models require more GVT computations to execute effectively. This means that the synchronization cost of the GVT computation is even more impactful for complex models. This is backed up by the fact that the asynchronous reduction scheme shows most improvement for the non-uniform models being tested. Additionally we see that the leash triggers optimum configurations even further limit the number of events executed per GVT, further explaining the performance difference between the two triggers.

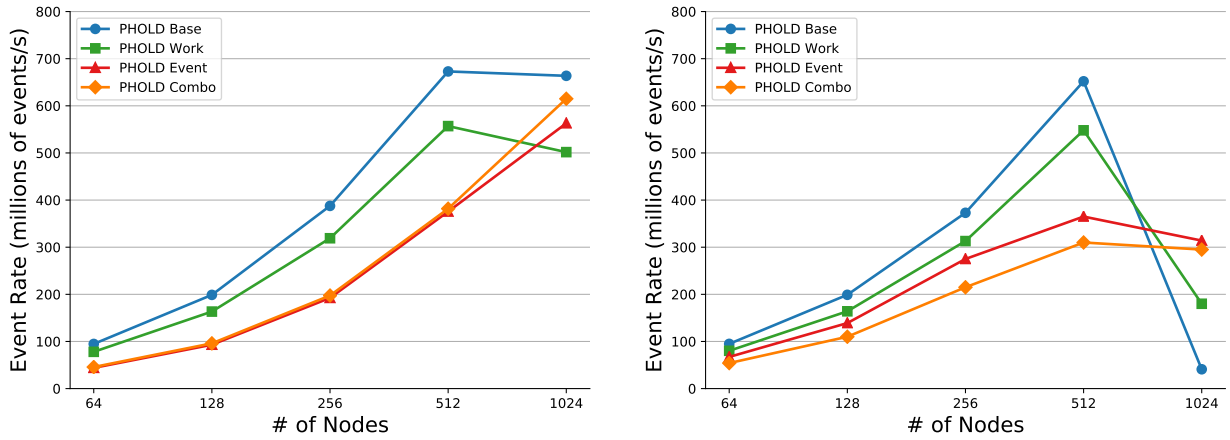

Figure 4.10: Strong scaling of the blocking algorithm with the count-based GVT trigger (left) and leash-based GVT trigger (right).

Scaling

Figure 4.10 shows the strong scaling performance of the PHOLD benchmark with the Blocking GVT algorithm by plotting event rate for a number of node counts. These runs use a larger version of the PHOLD benchmark which has 524,288 LPs as compared to the version with 131,072 LPs. We see that despite the leash trigger outperforming the count trigger in previous results, the count trigger shows better scalability as we scale up to larger node counts. For these experiments we used the best GVT configurations obtained from previous experiments on all node counts, and the lack of scalability in the leash version is due to the fact that the leash trigger inherently ties the number of GVT computations to the length of the simulation in virtual time, which does not change as we run on larger node counts. On the other hand, the count trigger ties the number of GVT computations to the number of events on each processor, which does change as we scale up the node count.

By looking at GVT frequency data in Figure 4.11, we see fundamental differences in how each trigger effects strong scaling. For the count-based trigger, each process will execute a set number of events every GVT cycle. By running the same model on more processes, the same number of total events are being spread across a larger number of processes. Therefore, not taking into account rollbacks, each process has fewer events to execute to complete the entire simulation. When using a count based trigger for the GVT, this results in fewer GVT computations as we scale up the number of nodes. For the leash trigger, GVT computations are computed every T units of virtual time, which does not change as the computation scales. Therefore we see roughly the same number of GVT computations as we scale up. The exception being that at the largest node counts, we see the number of computations

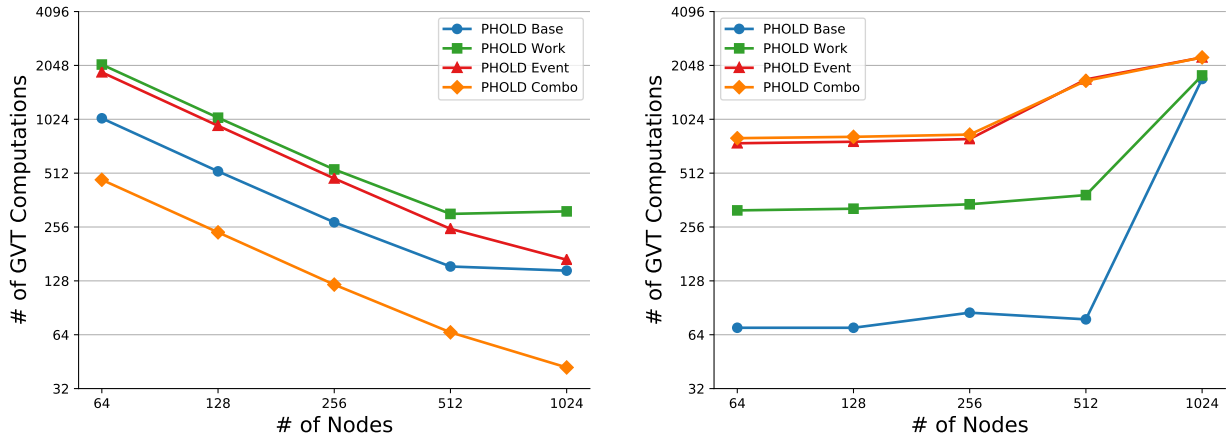

Figure 4.11: Strong scaling effects on the number of GVT computations for the count-based GVT trigger (left) and leash-based GVT trigger (right).

	Count Trigger				Leash Trigger			
Nodes	Base	Work	Event	Combo	Base	Work	Event	Combo
64	9,356	6,846	14,601	24,622	53,123	15,324	13,928	13,875
128	7,281	5,170	9,994	17,621	26,978	8,108	7,234	7,238
256	6,588	4,182	7,250	14,290	13,943	4,161	3,762	3,816
512	4,483	3,115	5,408	13,552	7,251	2,269	2,050	2,039
1,024	2,696	2,010	3,617	11,398	3,772	1,305	1,211	1,167

Table 4.7: The maximum number of events allocated at a given time across all processors for each node count and PHOLD configuration.

actually increase. This is due to sharp decreases in event efficiency. Figure 4.12 shows how this effects the average amount of time spent blocking event execution on each processor as we scale up. For the count trigger, the delay decreases fairly uniformly all the way up to the larger node counts. For the leash, the delay does not scale down as effectively due to the number of computations remaining roughly constant.

Furthermore, spreading events across more processes also effects the event efficiency of each simulation. At small scales, processes have a much larger pool of events to select from when choosing the next event to execute. As the node count increases, the events are spread across more processes resulting in fewer events per process to choose from. This is shown in Table 4.7, which shows the maximum number of events allocated on any one process in each run. At smaller node counts, the larger event pools result in processors being able to more accurately select events which will not be rolled back. At large scales, with each processor

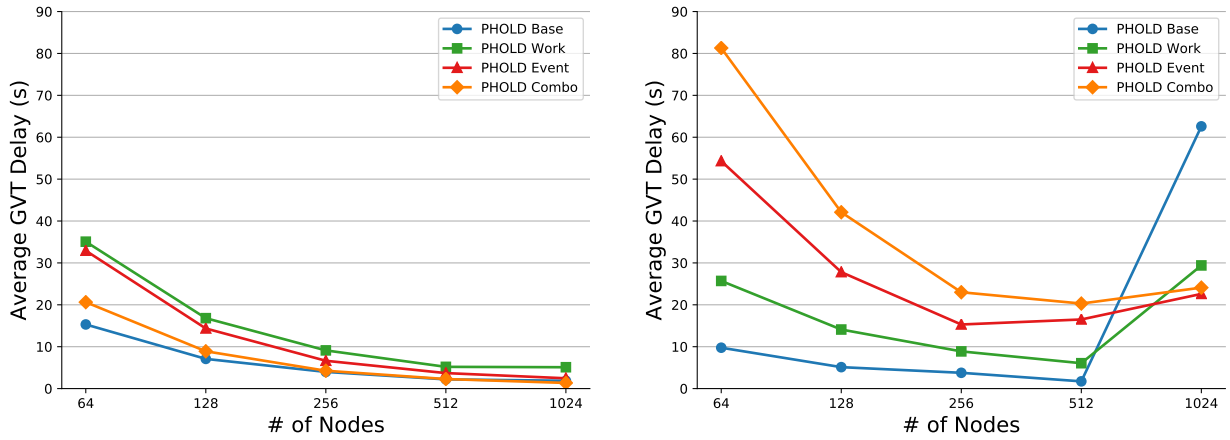

Figure 4.12: Strong scaling effects on the amount of time that event execution is blocking on the GVT computation for the count-based GVT trigger (left) and leash-based GVT trigger (right).

	Count Trigger				Leash Trigger			
Nodes	Base	Work	Event	Combo	Base	Work	Event	Combo
64	98%	98%	54%	54%	98%	96%	96%	95%
128	97%	98%	54%	54%	97%	95%	96%	94%
256	95%	96%	54%	53%	93%	94%	95%	93%
512	84%	85%	52%	51%	80%	88%	93%	92%
1,024	45%	41%	39%	41%	3%	25%	64%	63%

Table 4.8: Event efficiency for each node count and PHOLD configuration. Event efficiency generally decreases as the number of nodes increases.

having much fewer events, there is a higher likelihood that a processor will get stuck with events which are further in the future and more likely to be rolled back. This is shown in Table 4.8, and accounts for the scaling drop-offs we see as we run on larger node counts.

The addition of the asynchronous reduction reduces the synchronization costs at all node counts and achieves slightly better performance across the board, however scaling is largely similar. Figure 4.13 shows the event rates at each node count when the asynchronous reduction is turned on. The primary difference here is that there is that the drop-off in scaling when using the leash metric at larger node counts is not as sharp. This is due to the fact that the amount of time blocking became a much more significant portion of the simulation at large node counts for the leash trigger. Allowing event computation to resume earlier cut off some of that extra delay and also resulted in a higher event efficiency which

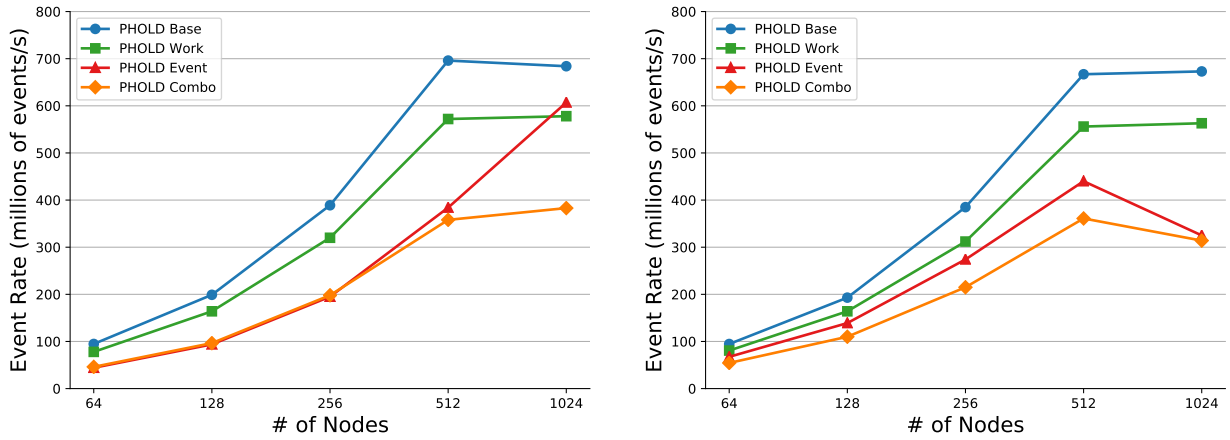

Figure 4.13: Strong scaling for the Blocking GVT algorithm when using the asynchronous reduction improvement from Section 4.2.2.

further curtailed the costs by lowering the number of GVT computations.

We have already discussed the sensitivity of this particular algorithm to the GVT frequency, and this effect is only exacerbated when scaling up to more nodes. In order to get better scaling with the Blocking algorithm, it is likely that the GVT frequency will have to be separately tuned for each node count, especially for the leash-based trigger.

4.2.3 Summary

While the basic Blocking GVT algorithm is still frequently used for high-performance simulations, it does incur a high synchronization cost, especially for complex models which require more frequent GVT computations to remain efficient. This synchronization cost is partially offset by the benefit of bounded optimism, which often has the side effect of keeping event efficiency relatively high. Using an asynchronous All-Reduce to finish the computation does allow us to eliminate some of the synchronization costs, however our experiments showed that the majority of the synchronization cost is from the QD portion of the algorithm instead. The algorithm is also sensitive to parameters which effect GVT computation frequency, and if not tuned properly can result in poor performance. This also affects scaling, where the parameters which are effective for one node count may not be effective for another. In the following sections we look at two algorithms that aim to eliminate synchronization cost entirely.

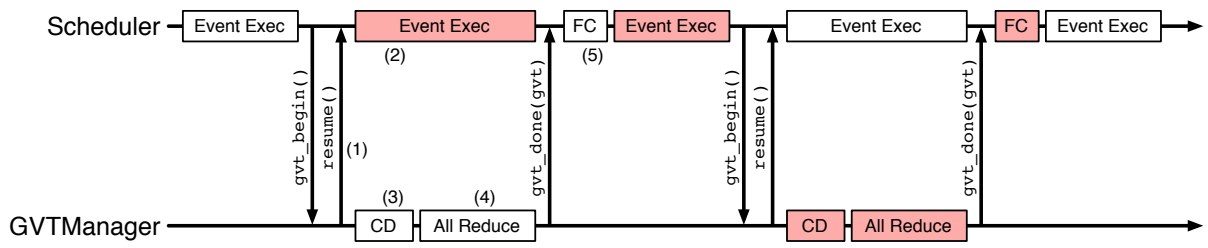

Figure 4.14: Diagram of the Phase-Based GVT algorithm. Both event execution and completion detection alternate between red and white phases in order to allow for continuous execution of events.

4.3 PHASE-BASED GVT ALGORITHM

The second algorithm 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 [19, 58, 20].

For this algorithm, we utilize sets of counters to monitor which events are in flight at any given time. A high-level diagram of the algorithm is shown in Figure 4.14. The algorithm runs in alternating phases (shown as red and white in the diagram), and tags events based on the phase during which they were sent. At some point, determined by the `Scheduler`, the `GVTManager` will switch phases. Upon doing so, it immediately resumes the `Scheduler` (1) which then continues to execute and send events in the new phase (2). During this time, the `GVTManager` uses a completion detection (CD) algorithm to wait for all events sent in the previous phase to be received (3). Once all events have been received it can use a reduction to estimate the new GVT (4) and inform the `Scheduler`. The `Scheduler` can then perform fossil collection based on the new GVT (5) and continue event execution until the next GVT computation. A more detailed description is given in the following paragraphs and follows the full algorithm description laid out in Algorithms 4.3 and 4.4.

4.3.1 Implementation

The Phase-Based GVT algorithm consists of two main pieces: event monitoring, and main control flow. Event monitoring is simply the portion of the algorithm which keeps track of the events in the simulation as they are being sent and received. Then, the main control flow describes how this information is utilized to actually compute a new GVT.

Algorithm 4.3 Phase-Based GVT algorithm event monitoring

```
1: function SendingEvent(event)          ▷ Called just before a remote event is sent
2:     event->phase = current_phase                          ▷ Tag the outgoing event
3:     sent[current_phase]++
4:     min_sent = min(event->ts, min_sent)   ▷ Keep track of the min sent timestamp
5: end function
6:
7: function ReceivingEvent(event)        ▷ Called after a remote event is received
8:     recvd[event->phase]++                        ▷ Update count based on events tag
9: end function
```

Event Monitoring

In order to correctly monitor and tag events, the `GVTManager` has to be made aware of the sending and receiving of events. In order to do so, the `GVTManager` has functions which are called just before sending and just after receiving events, shown in Algorithm 4.3. Lines 1 to 5 show that right before an event is sent, the `GVTManager` tags it with the current phase, updates its counter of sent events for the current phase, and keeps track of the minimum outgoing timestamp it has seen. This will be relevant later when determining each processors minimum timestamp. The `GVTManager` is also informed of received events on lines 7 to 9. The function is called after the event is received by its target LP, which means any causality violations have already been dealt with and the event is in the LPs pending event heap. At this point the events received counter is incremented based on the tag of the event, not the current phase.

Control Flow

Algorithm 4.4 shows the control flow for the rest of the Phase-Based GVT algorithm in detail. Like in the Blocking GVT algorithm, a trigger is required to determine when to switch phases and compute a new GVT. Both the count-based trigger and leash-based trigger described in the previous section can be used for this purpose. When it is time to compute a new GVT the `Scheduler` calls GVTBegin on its local `GVTManager`, which immediately calls `resume()` on the `Scheduler` (line 2). This allows the `Scheduler` to immediately resume executing events. As long as the `GVTManager` is ready to compute a new GVT it then switches to the next phase (line 7), which has two important consequences. First, any events sent from this point forward will be tagged with the new phase number (Algorithm 4.3 line 2). Secondly, when a new event is sent it will increment the sent counter for the new phase (Algorithm 4.3 line 3). This means that the sent counter for what is now the previous phase

Algorithm 4.4 Phase-Based GVT

```
 1: function GVTBegin()                                  ▷ Switch to next phase
 2:     scheduler->resume()
 3:     if ready then
 4:         ready = false
 5:         min_sent = MAX_TS
 6:         prev_phase = current_phase
 7:         current_phase = (current_phase + 1) % 2
 8:         sum_reduction(sent[prev_phase], recvd[prev_phase])
 9:     end if
10: end function
11:
12: function SumComplete(sent, recvd)     ▷ Called at completion of a sum reduction
13:     if sent != recvd then                  ▷ Repeat reduction until sent == recvd
14:         sum_reduction(sent[prev_phase], recvd[prev_phase])
15:     else
16:         lvt = min(min_sent, scheduler->min_time())
17:         min_reduction(lvt)
18:     end if
19: end function
20:
21: function MinComplete(new_gvt)          ▷ Called at completion of a min reduction
22:     current_gvt = new_gvt
23:     scheduler->gvt_done(current_gvt)
24:     ready = true
25: end function
```

will no longer be incremented on this processor. The last thing done when switching phases is to begin a reduction for summing the total number of sent and received events across all processors from what is now the previous phase (line 8).

Once the sum reduction completes, the runtime system calls the SumComplete method with the totals for sent and received events (line 12). If the counts are not equal, then we know that there are still some events that were sent during the previous phase that have not yet been received. In this case, we perform the same reduction again with the current values of the counters (line 14). Because the reduction is asynchronous and overlapped with other work, it is likely that more events have been received since the previous reduction began. Eventually, the two counters will be equal, at which point we know all events sent in the previous phase have been received. It is now time to estimate the new GVT. This is done by taking the minimum of our processors LVT and the smallest timestamp of any events we've sent since the most recent phase switch (line 16). A minimum reduction is then done

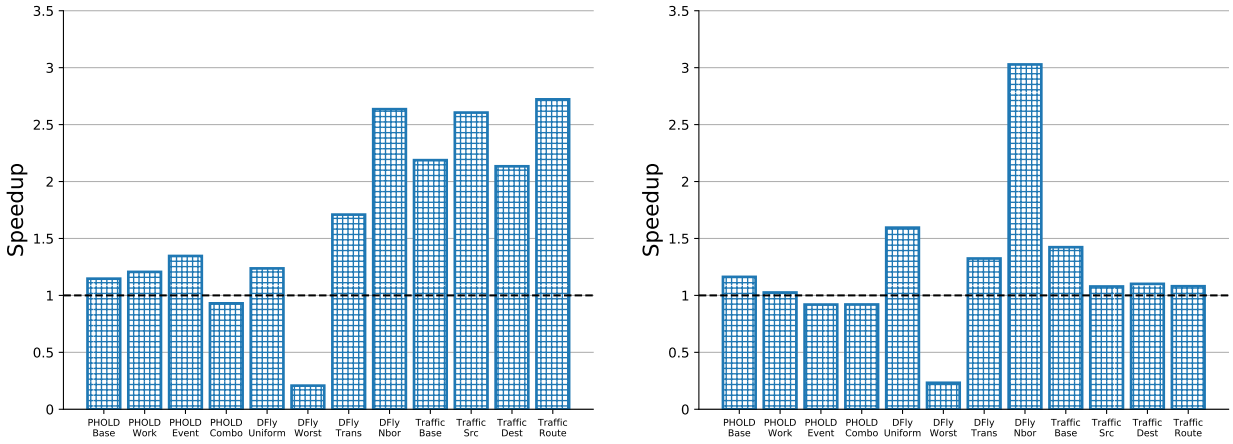

Figure 4.15: Speedup when using the Phase-Based GVT algorithm compared to the best Blocking GVT configurations for the count-based GVT trigger (left) and the leash-based GVT trigger (right).

to find the global minimum of these timestamps (line 17). The reason that the minimum sent timestamp must be taken into account is laid out in detail in [19]. In order to compute a valid GVT every event must be taken into account. The completion detection algorithm ensures that every event from the previous phase can be accounted for by the processors which received those events. However, the completion detection makes no guarantees about the locations of events which have been sent during the current phase. Since there is no guarantee that they have been received, we keep track of them on the sender side. By tracking the minimum of all events we have sent since a phase switch, we ensure that even if the smallest event in the system is still in flight it will be included in the GVT computation.

Once the minimum reduction is completed, the new GVT estimate is known, and it is passed along to the `Scheduler` (line 23). At this point the `GVTManager` is also marked as ready to begin another computation (line 24). The next time the `Scheduler` calls GVTBegin the process can be repeated to compute the next GVT. Throughout the entire execution, the runtime system is dynamically scheduling all communication and computation for the `GVTManager` alongside work for the `Scheduler` and `LPChares`. This allows the `Scheduler` to continue event execution throughout the entire simulation without ever needing to block for the GVT computation.

4.3.2 Performance Comparison

Figure 4.15 shows the performance of the Phase-Based GVT algorithm as speedup over the Blocking GVT algorithm from Section 4.2. Results are shown for both the count and

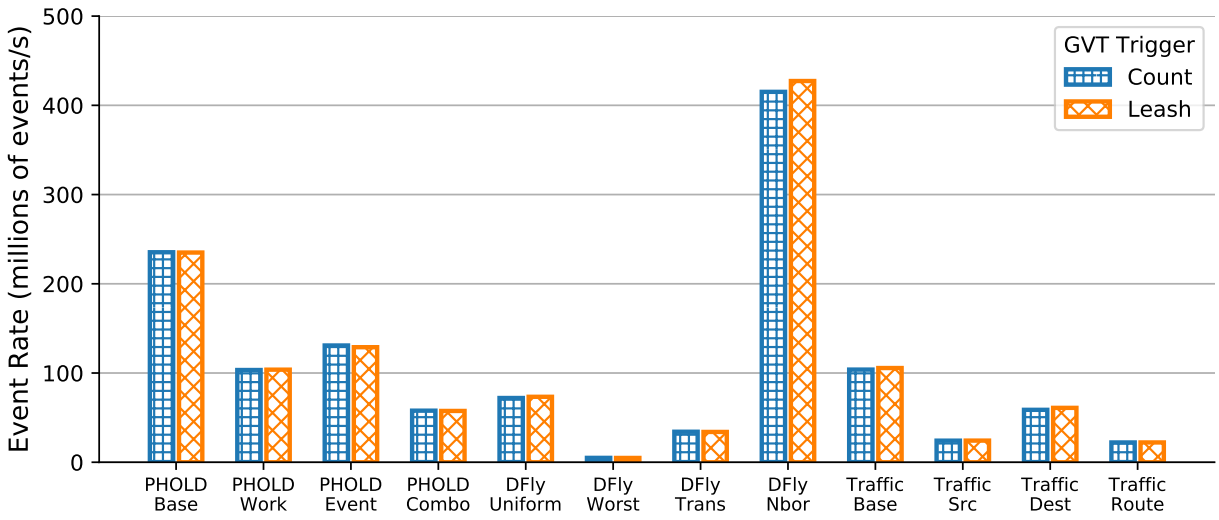

Figure 4.16: Event Rate of the Phase-Based GVT algorithm for each model configuration under the count-based and leash-based GVT triggers.

leash trigger. We see significant speedup in many of the configurations when comparing the count-based trigger. When looking at the results for the leash trigger, the improvements are not as significant. However, if we compare the event rates of the Phase-Based algorithm under each trigger we can clearly see why. Figure 4.16 shows the event rates for the algorithm using both triggers and we see that they are identical. The larger speedups over the count-based trigger are merely due to the fact that the count-based trigger had a lower event rate than the leash trigger for the Blocking algorithm. Since the Phase-Based GVT algorithm does not block event execution, the trigger that determines when to compute the GVT has little effect on the performance of event execution.

As mentioned in the previous section, the Blocking algorithm has the additional side-effect of bounding the optimism of the simulation by preventing LPs from getting too far ahead of their slower counterparts. By removing the bound on optimism, event efficiency under the Phase-Based algorithm is now entirely based on model characteristics and the layout of LPs to processors. Looking at Table 4.9, which shows event efficiencies of each configuration under the Blocking and Phase-Based algorithms, we see that event efficiency is lower across the board when using the Phase-Based algorithm. By removing the bound on optimism from the GVT algorithm, LPs which advance far into the future cause more frequent rollbacks, and therefore more overhead via reverse computation, anti-events, and event re-execution. We also see that models which achieved the highest speedups in Figure 4.15 correspond to the models whose event efficiencies are highest, and therefore closer to what they were under the Blocking algorithm. Despite the fact that event efficiency is lower, we see at least some

	PHOLD				Dragonfly				Traffic			
GVT Algorithm	Base	Work	Event	Combo	Uniform	Worst	Trans	Nbor	Base	Src	Dest	Route
Blocking	98%	76%	84%	93%	74%	39%	85%	68%	96%	97%	96%	97%
Phase-Based	95%	52%	60%	31%	36%	2%	27%	67%	55%	16%	52%	15%

Table 4.9: Event efficiency of the Phase-Based GVT algorithm compared to the event efficiency of the best Blocking GVT algorithm configuration.

speedup in almost all cases due to the fact that the synchronization cost of blocking event execution is almost entirely removed.

Frequency and Overhead Analysis

In the previous section, we showed how the GVT frequency had a large impact on performance and coupled together the notions of event efficiency and synchronization cost. This meant that it was important to tune the GVT to get the best performance, and the parameters used depended heavily on the model being run. It also made scaling difficult because parameters would need to be tuned independently for each node size. All of this was largely due to the fact that the GVT algorithm blocked event execution while computing the next GVT. For the algorithm presented in this section, event execution is not blocked by the GVT algorithm, and the runtime system adaptively overlaps the work of the `Scheduler` and the `GVTManager`. Because of this, both event efficiency and synchronization cost are not determined by GVT frequency. The amount of time spent blocking event execution is virtually zero regardless of frequency, and event efficiency is dependent on model characteristics and LP layout.

As a result, changing the frequency of the GVT computation for the Phase-Based GVT algorithm had little to no effect on the performance of each model tested. By running a wide variety of intervals for each trigger, the maximum difference in both event rate and event efficiency of each model was approximately 1% in almost all cases. The main effect seen by adjusting frequency was that a less frequent GVT computation would result in more event memory being used, with simulations possibly crashing from running out of memory at the extreme end of the spectrum. For more complex models, like Traffic and Dragonfly, it was possible for memory consumption to out-pace the GVT computation, making it very difficult to get a configuration which would run to completion on machines with less memory such as Vesta and Mira.

In terms of overheads, the primary cost of the Phase-Based algorithm is in the amount of collective communication it requires to complete. There are potentially multiple successive

	PHOLD			
	Base	Work	Event	Combo
# of GVT Computations	945	991	1,298	1,592
# of GVT Reductions	3,887	4,270	5,553	6,890
Reductions per GVT	4.11	4.31	4.28	4.33

Table 4.10: The number of GVT computations and All-Reduces required for each PHOLD configuration.

All-Reduce calls required as the algorithm waits for the arrival of messages from the previous phase. In a perfect world, each computation would require a single All-Reduce, where each processor reaches the All-Reduce at the exact same time and all events have already been sent and received. A slightly more reasonable hope would be two All-Reduce calls per computation. One that signifies that every processor has switched phases, and then one more that comes after all events have been received. Table 4.10 shows the number of GVT computations and reduction calls for each PHOLD model configuration. As we can see, the number of calls per GVT in this case ends up being more than four in each case. As we will show in the next section, if the GVT algorithm is made virtual time aware and more adaptive in how it uses the reductions, we can cut down the amount of communication required to compute a GVT.

Scaling

Figure 4.17 shows strong scaling for each PHOLD configuration using the Phase-based GVT algorithm. We see good scaling maintained even up to the larger node counts, and there is much less of a dropoff than with the Blocking GVT algorithm scaling from Figure 4.10. Scaling is especially good for the configurations with an unbalanced distribution of events (PHOLD Event and PHOLD Combo). In many cases, we actually see super-linear speedup up until the largest node counts.

The fact that the GVT is run without blocking event execution helps scalability in this case. As we saw with the leash triggered Blocking algorithm, the time spent blocking execution on the GVT computation caused a scaling bottleneck. Even with the count trigger, the amount of time spent blocking event execution did not always scale effectively which resulted in bottlenecks at higher node counts. For the Phase-based algorithm, event execution is never blocked which results in zero GVT delay at all node counts. The only hindrance

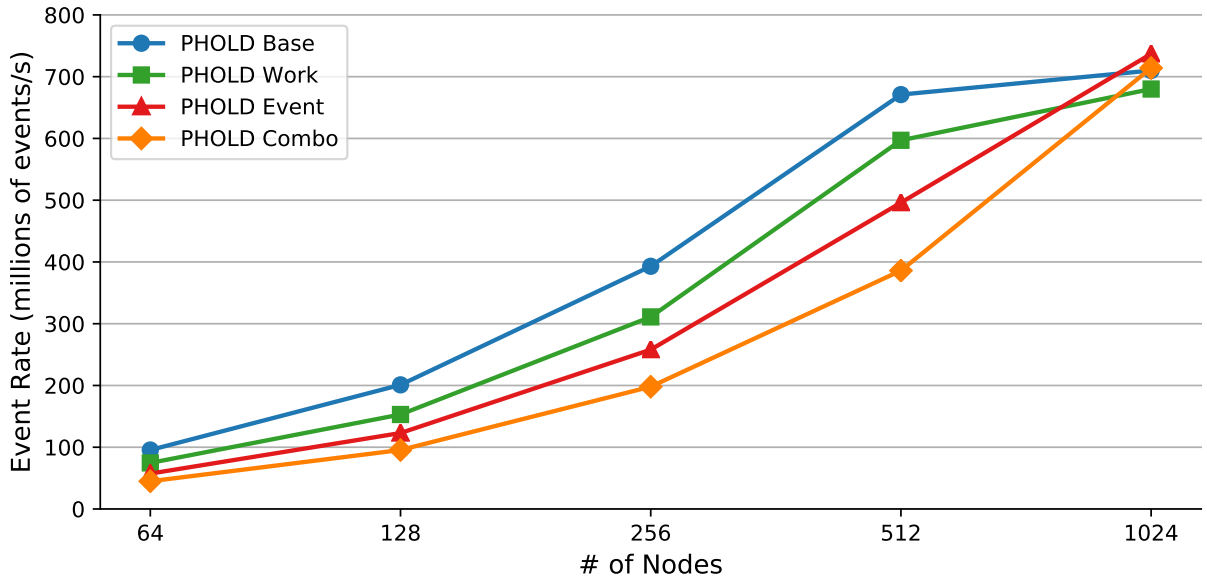

Figure 4.17: Strong scaling of the Phase-Based GVT algorithm.

to event execution is the overhead of communication required by the GVT algorithm. Figure 4.18 shows the number of GVT computations and the number of All-Reduces required at each node count. We see near-linear scaling for both quantities as the number of nodes increases.

4.3.3 Summary

The fully asynchronous Phase-Based GVT algorithm studied in this section completely removes the need to block event execution that is present in the Blocking GVT algorithm from the previous section. For some models, this drastically improves performance, especially for those which previously spent a large fraction of their execution time waiting on the GVT computation. However, once again we see a tradeoff between event efficiency and synchronization cost, due to the fact that removing blocking also removes the bounded optimism which previous kept event efficiency high. This limits the effectiveness of this algorithm especially for models which behave poorly when allowed to execute freely. Furthermore, this algorithm can be tricky to tune and often incurs large amounts of collective communication due to the fact that it is completely unaware of virtual time and event rollbacks, which can cause little progress to be made for a given GVT computation. In the next section, we present an algorithm which uses similar principles as the Phase-Based approach to eliminate synchronization costs, but which is also cognizant of virtual time and its effect on the GVT

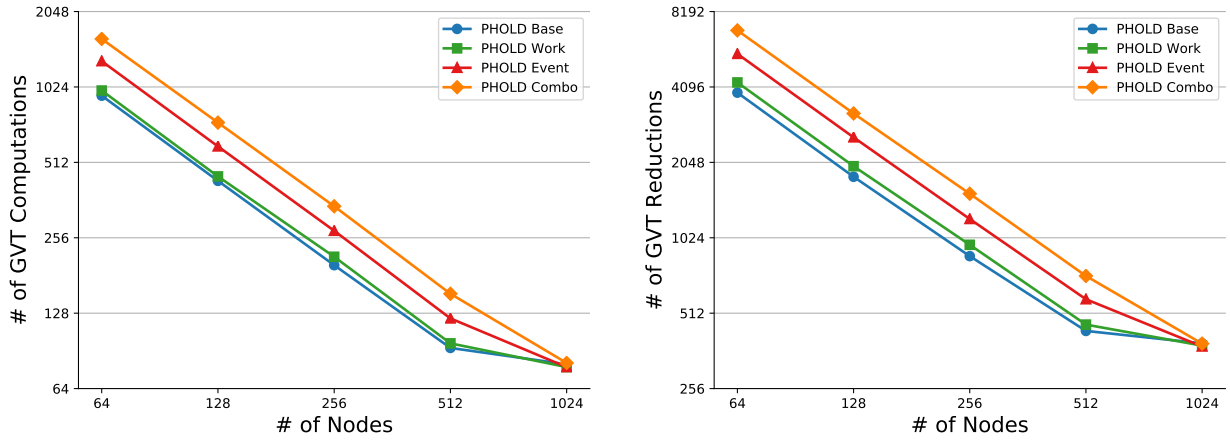

Figure 4.18: Strong scaling effects on the number of completed GVT computations (left), and the number of All-Reduce calls require to complete them (right) for each PHOLD configuration. There are roughly four reductions required per GVT computation.

computation. Furthermore, we look at techniques to improve event efficiency independently from the synchronization, both in the next section and in Chapter 5.

4.4 ADAPTIVE BUCKETED GVT ALGORITHM

The third algorithm we will examine is also a completely non-blocking GVT algorithm, but attempts to improve on the Phased-Based GVT algorithm in a few ways. It is still based upon the idea of a completion detection algorithm which uses a sent/received counter to wait for a set of events to be received. However, it also uses the extra event dependencies present in PDES to more effectively guide the algorithm. We will refer to it as the Adaptive Bucketed GVT algorithm because in order to remain non-blocking, it divides virtual time into buckets which each maintain independent event counts. The number of buckets included in each GVT is adaptive and based on the speed at which processors are progressing through virtual time.

Figure 4.19 shows how a single processor splits up virtual time into buckets. In this diagram, the current bucket based on the current LVT of the processor is colored in yellow. 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

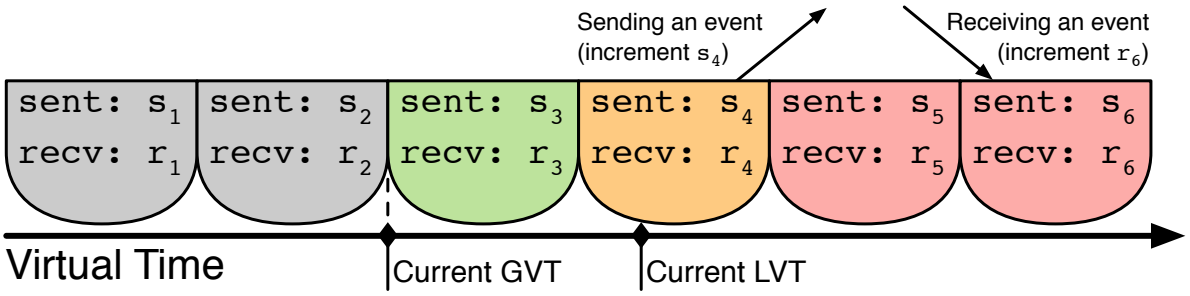

Figure 4.19: Diagram of bucketing done by the Adaptive Bucketed GVT algorithm.

Algorithm 4.5 Adaptive Bucketed GVT algorithm event monitoring

```
1: function SendingEvent(event)          ▷ Called just before a remote event is sent
2:     sent[event->ts / bucket_size]++
3:     AttemptGVT()
4: end function
5: function ReceivingEvent(event)        ▷ Called after a remote event is received
6:     recvd[event->ts / bucket_size]++
7:     AttemptGVT()
8: end function
```

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 increase. Because of this, the received counter that is incremented will always be in the current bucket or a future bucket. The event monitoring pseudocode is shown in Algorithm 4.5 and is similar to the monitoring done in the Phase-Based GVT algorithm, although it does not require outgoing events to be modified.

4.4.1 Implementation and Correctness

The full algorithm for how these buckets are used to compute the GVT is laid out in detail in Algorithm 4.6. As events are sent and received on each processor, or when the `Scheduler` calls GVTBegin, the `GVTManager` will check to see if it has passed any buckets

beyond the current GVT by calling the AttemptGVT function. If it has, it will contribute the number of buckets passed to an asynchronous All-Reduce (line 10). Once all processors have contributed, each `GVTManager` 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 its summed sent counter matches its summed received counter, it has been passed by all processors, and all preceding buckets are also completed. The number of completed buckets are counted up in lines 21 to 27. Once we know how many buckets have been completed, the GVT can be updated to the edge of the latest completed bucket (lines 28 to 31). 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 32 to 37).

One difference between this algorithm and the previous two is that the `GVTManager` can operate almost completely independently from the `Scheduler`. The `GVTManager` monitors sent and received events, and can detect on its own when to start computing the next GVT. This also means that the previous notion of GVT trigger is no longer relevant for this algorithm. The `Scheduler` does not affect the GVT frequency at all, and the calls to GVTBegin are only required to make sure the `GVTManager` checks for progress in cases where no events are being sent or received, ie at the end of a simulation. The second difference is that, unlike with the Phase-Based algorithm, the Adaptive Bucketed algorithm is able to adjust effectively to rollbacks. The Phase-Based algorithm must have a single transition between one phase and the next so that once the reductions begin there is a guarantee that the sent event counters will not change. This can result in very small GVT advances, and therefore more required GVT computations, for cases where heavy rollbacks occur after phase switches. As we'll show later in this section, this can also result in more communication required to compute the GVT with the Phase-Based algorithm compared to the Adaptive Bucketed algorithm. In the Adaptive Bucketed algorithm, extra information about event causality, as well as the minimum number of buckets being included in the reduction, allows for both the sent and received counters to be incremented during the GVT computation, as well as ensuring a minimum one bucket of progress per completed GVT computation. This results in both fewer reductions, and a more predictable total number of GVT computations due to the fact that the total number of complete GVTs is bounded by the number of buckets.

Algorithm 4.6 Adaptive Bucketed GVT

```
 1: function GVTBegin()
 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 MinComplete(b)
15:     lvt = scheduler->min_time()
16:     new_buckets = (lvt - current_gvt) / bucket_size
17:     reduction(sent[b],recvd[b], new_buckets)
18: end function
19:
20: function ReductionComplete(sent[b], recvd[b], new_buckets)
21:     for x = 0; x < min(b, new_buckets); x++ do
22:         if sent[x] == recvd[x] then
23:             completed++
24:         else
25:             break
26:         end if
27:     end for
28:     if completed > 0 then
29:         current_gvt += completed * bucket_size
30:         scheduler->resume()
31:     end if
32:     if new_buckets - completed > 0 then
33:         residual = new_buckets - completed
34:         lvt = scheduler->min_time()
35:         new_buckets = (lvt - current_gvt) / bucket_size
36:         reduction(sent[residual],recvd[residual], new_buckets)
37:     end if
38: end function
39:
```

Proof Sketch

The crux of the Adaptive Bucketed GVT algorithm is marking buckets as complete in order to advance the GVT. Correctness of the algorithm entirely relies on the guarantee that no more events will ever be created that fall within a completed bucket. This is due to the fact that the GVT is always at the end of a completed bucket, and therefore all events that are committed belong to completed buckets. If an event were to arrive within a completed bucket, the simulator would not be able to perform the requisite rollbacks and could not correctly complete the simulation. Therefore, to prove that the Adaptive Bucketed algorithm always computes a valid GVT estimate, we need to show that no event will ever arrive within a bucket which has been marked complete. In the following paragraphs we sketch a basic proof by contradiction to show that this assumption holds true.

First, assume that bucket b has been marked as complete by the algorithm. To be marked as complete, b must meet all three of the conditions for marking a bucket complete: the buckets counters are equal, all processor LVTs have passed the bucket, and all previous buckets are completed. Note that bucket b was marked complete immediately after the completion of a reduction in Algorithm 4.6 line 23, where these three conditions are checked. Now assume there is some event, e, sent by some some processor, p, and that e falls within bucket b. This event was sent either before or after p contributed to the reduction which ultimately resulted in b being marked as complete.

If e was sent before the contribution, then it would have been included in the count of sent events. However, since it was received after the reduction completed, it was not included in the count of received events. Since all events are received after they are sent, the global count of sent events is always greater than or equal to the number of received events. As such, since e was included in the sent count, but not the received count, it is impossible for the sent and received count for bucket b to be equal in that reduction and therefore b would not have been marked as complete.

If instead e was sent after the contribution, it was not included in the sent or received counts and therefore it is possible that those counts were equal at the completion of the reduction. However, e must have been created by some event e', which has a timestamp less than or equal to that of e. If e' existed in a pending queue before the contribution, then the LVT of the processor where e' existed would have had an LVT that was at least as small as the timestamp of e', which has not exceeded b. Therefore b would not be marked as completed.

If e' did not exist in a pending queue when the contribution was sent, then we can trace back the chain of events which caused it to appear in the pending queue (either by creating

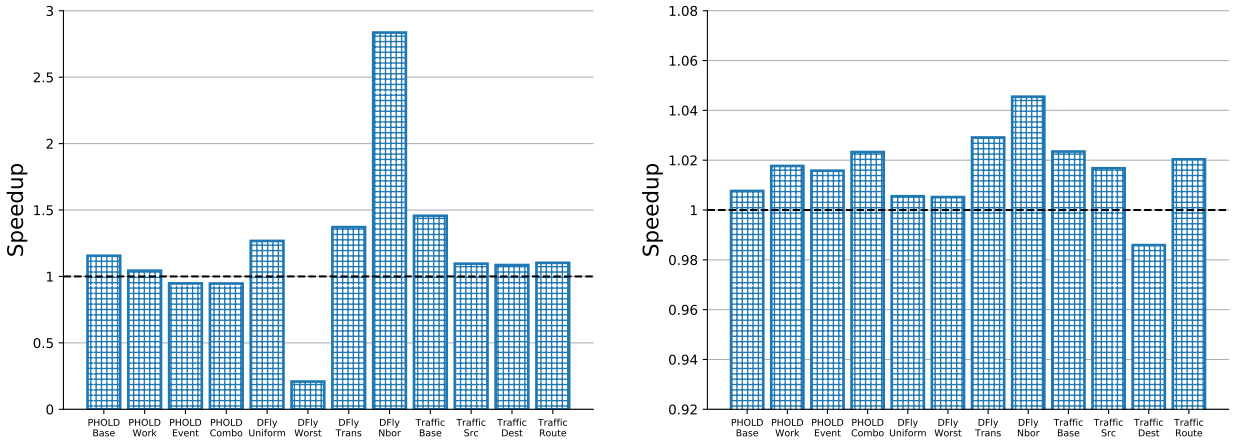

Figure 4.20: Speedup of the Adaptive Bucketed GVT algorithm over the best Blocking GVT configuration (left) and the best Phase-Based GVT configuration (right).

	PHOLD				Dragonfly				Traffic			
GVT Algorithm	Base	Work	Event	Combo	Uniform	Worst	Trans	Nbor	Base	Src	Dest	Route
Blocking	98%	76%	84%	93%	74%	39%	85%	68%	96%	97%	96%	97%
Phased	95%	52%	60%	31%	36%	2%	27%	67%	55%	16%	52%	15%
Bucketed	96%	53%	61%	31%	36%	2%	28%	68%	55%	16%	53%	15%

Table 4.11: Event efficiency of each model configuration for the three different GVT algorithms.

it or causing it to be rolled back). Either way, these events have a timestamp less than or equal to that of e' and we can apply the same logic we used for e to show that either the counts in the reduction will be unequal or that the LVT has not exceeded b. We therefore have a contradiction: if event e exists within bucket b, the bucket b could not possibly have been marked as completed.

4.4.2 Performance Comparison

Figure 4.20 shows the event rate of the Adaptive Bucketed GVT algorithm plotted as speedup over the best Blocking GVT algorithm configuration (left), and the best Phase-Based GVT configuration (right). In each simulation except for Traffic Dest we see slight speedups over the Phase-Based GVT algorithm, and correspondingly the speedups we see over the Blocking algorithm are similar to those we saw for the Phase-Based algorithm. This means that in all but two PHOLD configurations and one Traffic configuration, the Adaptive Bucketed algorithm provides the best performance. As further reinforcement of what we demonstrated in the previous section, the event efficiency of each model under the Adaptive Bucketed algorithm is nearly identical to the efficiencies under the Phase-

	PHOLD				Dragonfly				Traffic			
Bucket Size	Base	Work	Event	Combo	Uniform	Worst	Trans	Nbor	Base	Src	Dest	Route
# of Comps												
1	1,013	1,021	1,016	1,024	773	678	2,707	99	806	1,022	860	1,019
4	256	256	256	256	734	642	1,818	90	247	256	255	256
8	128	128	128	128	454	646	1,015	81	127	128	128	128
Buckets per Comp												
1	1.01	1.00	1.00	1.00	10.59	12.08	3.02	82.74	1.27	1.00	1.19	1.00
4	1.00	1.00	1.00	1.00	2.79	3.19	1.12	22.75	1.03	1.00	1.00	1.00
8	1.00	1.00	1.00	1.00	2.25	1.58	1.00	12.64	1.00	1.00	1.00	1.00

Table 4.12: The number of completed GVT computations for each of three different bucket sizes, as well as the average number of buckets per computation.

	PHOLD				Dragonfly				Traffic			
Bucket Size	Base	Work	Event	Combo	Uniform	Worst	Trans	Nbor	Base	Src	Dest	Route
# of Redns												
1	2,005	2,024	2,011	2,040	774	681	2,724	100	1,276	1,965	1,350	2,027
4	512	512	513	512	784	656	2,686	91	448	512	507	512
8	260	258	257	256	782	670	1,938	82	251	256	262	256
Redns per Comp												
1	1.98	1.98	1.98	1.99	1.00	1.00	1.00	1.01	1.58	1.92	1.57	1.99
4	2.00	2.00	2.00	2.00	1.06	1.02	3.47	1.01	1.82	2.00	1.99	2.00
8	2.03	2.02	2.01	2.00	1.72	1.03	1.90	1.01	1.98	2.00	2.05	2.00

Table 4.13: The total number of reductions done by the GVT algorithm for each of three different bucket sizes as well as the number of reductions done per completed GVT computation.

Based algorithm (Table 4.11). This again shows that without bounded optimism the event efficiency becomes totally dependent on the characteristics of the model being executed.

Bucket Size Analysis

As with the previous algorithms, we also want to analyze how much effect GVT frequency has on the Adaptive Bucketed GVT algorithm, as well as which characteristics of the algorithm it actually affects. As previously mentioned, the GVT frequency is no longer dictated by the `Scheduler`, and is instead directly related to the size of the buckets. Because the GVT is computed as processors pass bucket boundaries, increasing the size of buckets will result in longer times between GVT computations. However as mentioned previously, the algorithm does have some flexibility to adapt by advancing multiple buckets in a single GVT computation. Table 4.12 shows the total number of completed GVT computations for each model configuration for 3 different bucket sizes. In this case, we measure a completed GVT computation as the number of times the `Scheduler` is informed of a new GVT. This is also equivalent to the number of times fossil collection is executed. Because of the way the algorithm adapts to simulation conditions, this does not necessarily correspond to the

number of times a GVT computation is started, nor does it necessarily correspond directly to the number of reductions required. Table 4.12 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 `GVTManager` 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 4.6 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 4.13, 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. The Phase-Based GVT algorithm from the previous section required on average more than four reductions per GVT computation, and used a similar number of GVT computations as the Adaptive Bucketed algorithm with a bucket size of one. This results in a fairly dramatic decrease in the amount of communication required by the Adaptive Bucketed algorithm when compared to the Phase-Based algorithm. This is due to the fact that the Adaptive Bucketed algorithm is able to adaptively adjust the size of the GVT as the algorithm runs. It is also virtual time aware, which allows it to more gracefully handle rollbacks.

Looking at Figure 4.21, we see that this algorithm is also fairly resilient to frequency changes. Changing bucket size has a small impact on the overall performance, which takes a bit of the burden of tuning the GVT algorithm off of the programmer. In fact, the ratio

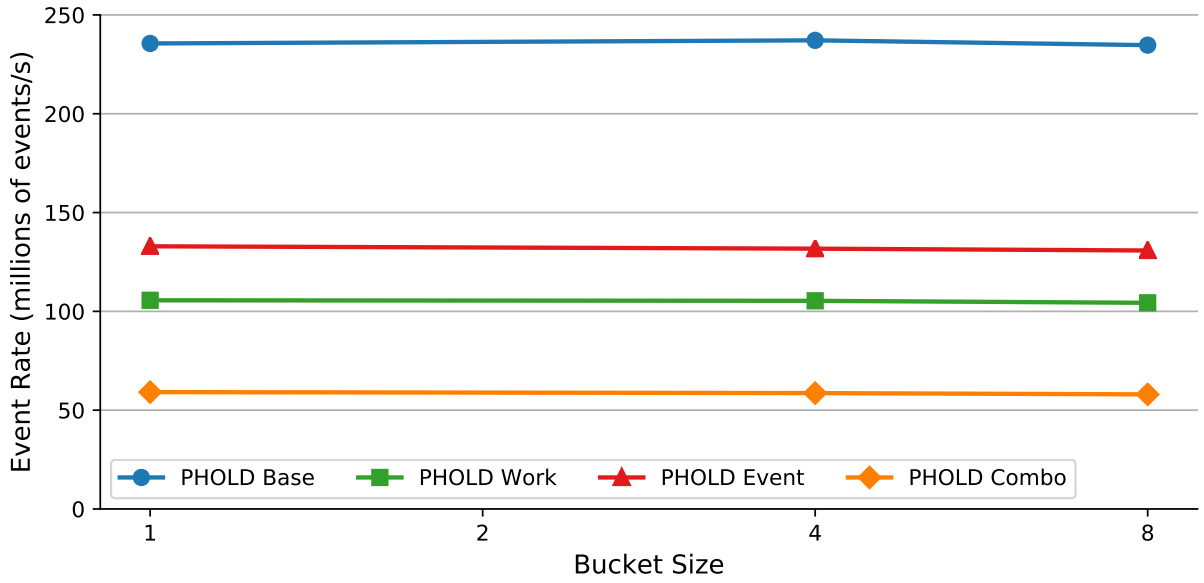

Figure 4.21: Event rates for each PHOLD model configuration at various bucket sizes. Event rate differs by less than 2%.

between the best performing configuration and worst performing configuration for all models did not exceed 1.02 in any of the experiments run. Not shown are the event rates for the Dragonfly and Traffic models, however they show the same results as we see for PHOLD. It is also important to note that completed GVT computations in this algorithm are guaranteed to advance by at least one bucket. This guarantee did not exist for the Phase-Based GVT algorithm which could sometimes be tricky to tune when event memory consumption was high. Computing it infrequently could result in too many events getting created before fossil collection. However, even if computing it very frequently, it was possible to run into situations where a GVT computation resulted in a very small GVT advance due to rollbacks. During the time that the GVT was computing, execution of events continued, and could still out pace fossil collection. Since the Adaptive Bucketed GVT is able to adapt to rollbacks by cutting down on the region of time being considered by the computation, it is able to update the GVT while still continuing to check for completion of later buckets. Even if rollbacks result in no buckets being valid, it just cuts the computation short and is immediately ready to start a new GVT. The Phase-Based algorithm always runs to completion, which may delay subsequent GVT computations.

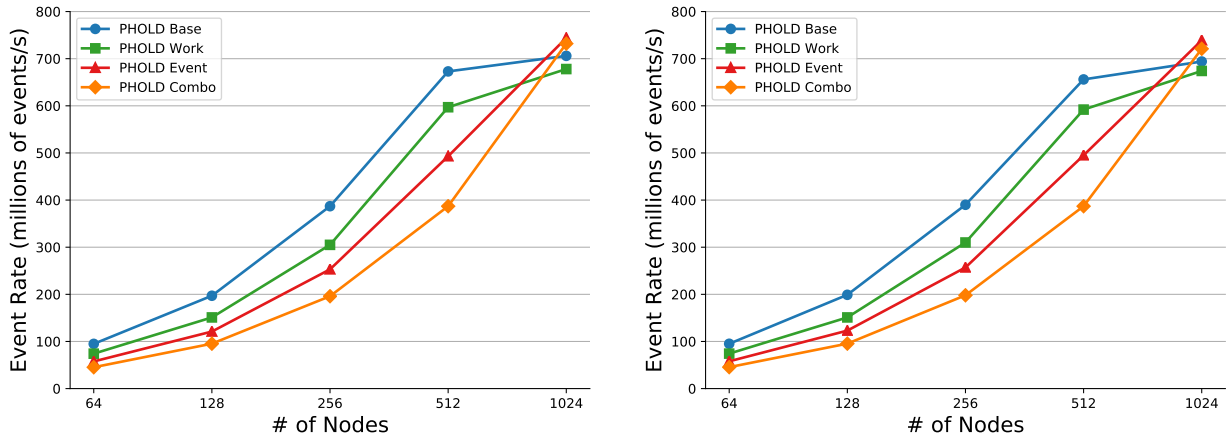

Figure 4.22: Strong scaling of the Adaptive Bucketed GVT algorithm with bucket sizes of one (left) and four (right).

Scaling

Like with the Phase-based GVT algorithm in the previous section, the Adaptive Bucketed GVT algorithm shows favorable scaling on the PHOLD benchmark due to the lack of blocking event execution. As shown in Figure 4.22, it shows effective scaling up to the max number of nodes tests. It scales more effectively than the Blocking GVT algorithm from Section 4.2, and slightly better than the Phase-based algorithm from Section 4.3. For PHOLD Combo, we achieve 16.17× speedup when comparing the 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 4, 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 either GVT algorithm competes with event execution is contention for communication resources. Figure 4.23 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

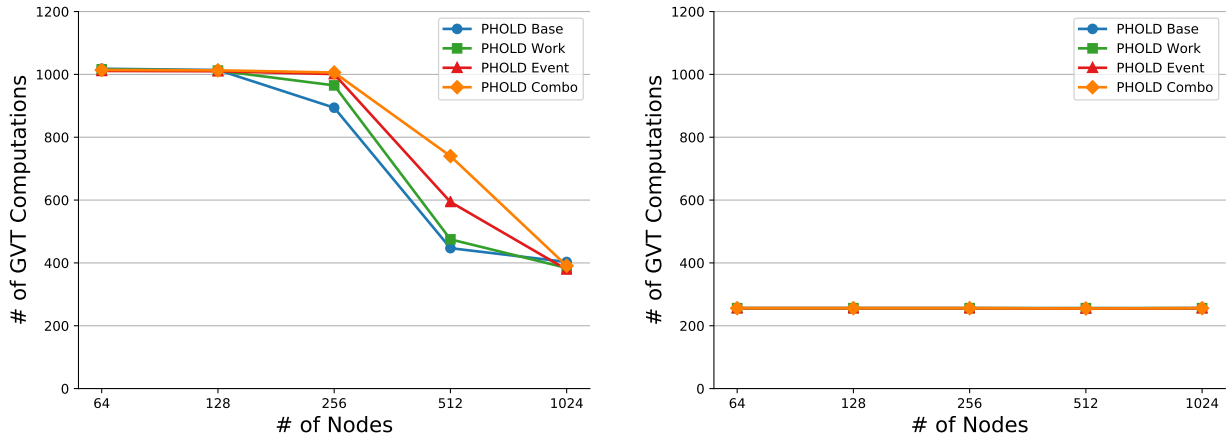

Figure 4.23: Strong scaling effects on the number of completed GVT computations for each PHOLD configuration for bucket sizes of one (left) and four (right).

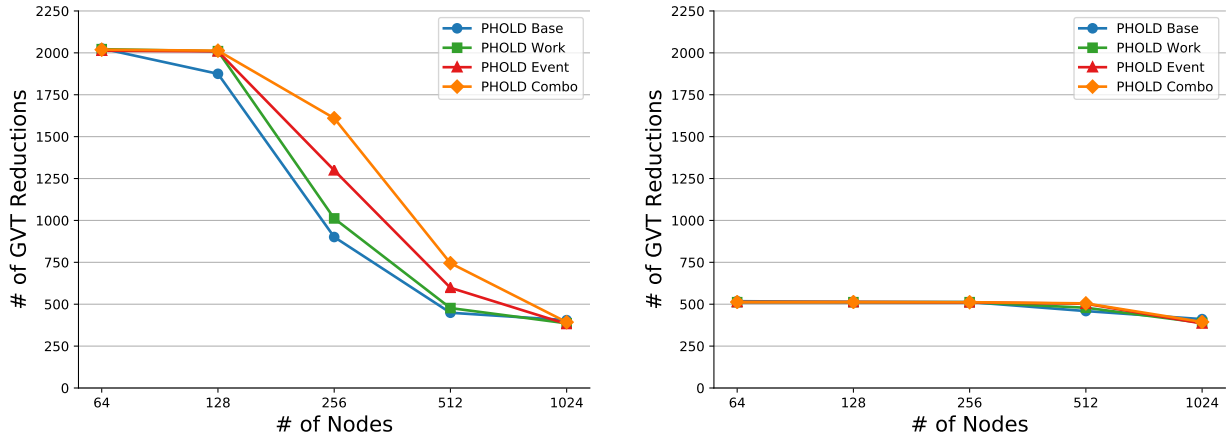

Figure 4.24: Strong scaling effects on the number of All-Reduce calls for each PHOLD configuration for bucket sizes of one (left) and four (right).

directly results in less communication as shown in Figure 4.24, which shows the total number of All-Reductions done by the `GVTManager` 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. 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.

4.4.3 Adaptive Event Throttling

One major downside of both the non-blocking algorithms we have explored is the sharp decrease in event efficiency when compared to the Blocking GVT algorithm. The Blocking algorithm bounded the optimistic execution of each model as a side-effect of halting event execution to compute the GVT. As we saw in Section 4.2, this tightly coupled event efficiency to synchronization cost which made tuning the GVT frequency critical to getting the best overall performance. Furthermore, it often meant to get high performance, we required a higher synchronization cost. Now that we have completely eliminated the blocking cost from the GVT algorithm, we would still like another method for improving event efficiency that is not coupled to GVT frequency or synchronization cost. Due to its already introspective nature, the Adaptive Bucketed GVT algorithm provides the perfect framework for such a method.

Our method, inspired by observations from the Blocking algorithm as well as ideas used in the SPEEDES system developed by Steinman [11, 18], revolves around the fact that many rollbacks come about due to the fact that certain processors get too far ahead in virtual time compared to others. It is also motivated by the fact that anti-events can be far more expensive than 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. 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]. However, there are a few key differences with our approach. 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 throttling 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 throttle events based on the risk estimated for each event independently. It may stop certain events, while not stopping others.

In order to throttle outgoing events, we use the fact that events are already passed through

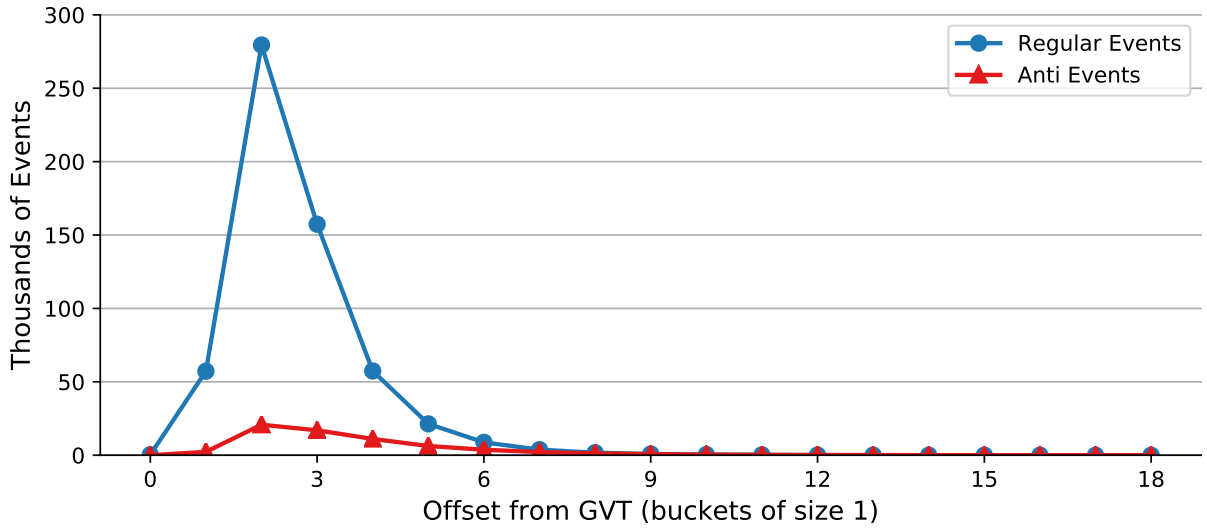

Figure 4.25: The number of regular events and anti-events at each offset from the current GVT (in buckets) for PHOLD Combo with a bucket size of one.

the `GVTManager` for the purpose of correctly updating bucket counts. With only minor changes, we can use this data for more than just computing the next GVT. As events pass through the `GVTManager`, the `GVTManager` 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 `GVTManager`. If the `GVTManager` 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 `GVTManager` 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 `GVTManager` 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

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

Table 4.14: Average lag between an event and its corresponding anti-event (in buckets).

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.

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 `GVTManager` which monitors events and anti-events based on their offset from the GVT at the time they are sent. Figure 4.25 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 `GVTManager` 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 sent, events in the same hash bucket

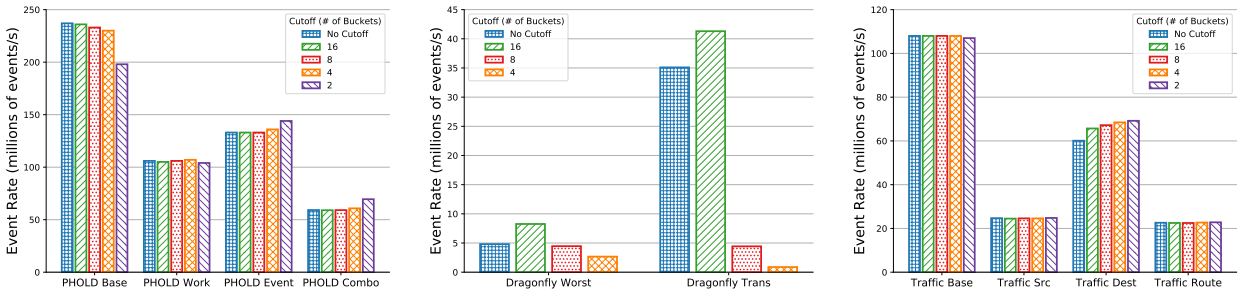

Figure 4.26: Event rates for each model when throttling event sends based on their offset from the current GVT. The higher the offset cutoff, the further from the GVT an event has to be to get throttled. Therefore bars are arranged by increasing aggressiveness in terms of buckets that will be throttled.

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 `GVTManager` 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 can not properly progress. In order to determine which events to release, we also rely on data collected by the `GVTManager` to get some intuition about how cancellations work. Table 4.14 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 previously, holding events too long may hurt performance by causing rollbacks when we eventually do release them and they reach their destinations late.

To see how these changes affect performance, we plot event rate of each model using various offset cutoffs in Figure 4.26. The figure shows event rates for cutoffs of 2, 4, 8, and

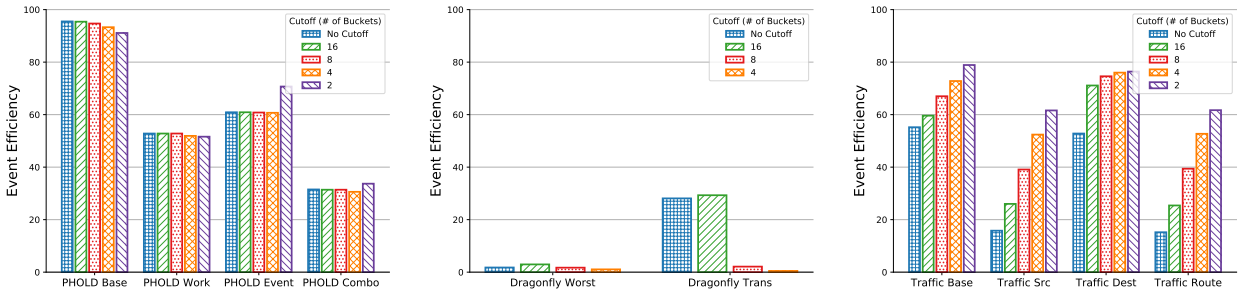

Figure 4.27: Event efficiency for each model when throttling event sends based on their offset from the current GVT.

16 buckets, as well as the event rates where no events are held. The bars are arranged from most conservative to most aggressive in terms of number of events held. We see varying performance based on configuration. For PHOLD Base, which is extremely uniform and already has a very high event efficiency, throttling events actually hurts performance. For each other PHOLD configuration, throttling improves event rate by 1% for PHOLD Work up to an 18% improvement for PHOLD Combo. The two Dragonfly configurations shown see large speedups due to both models suffering heavily from a low event efficiency. Dragonfly Worst gets 1.75× speedup when conservatively throttling events. Unlike PHOLD, the results for Dragonfly actually get worse as throttling becomes more aggressive. As we will show later, throttling too aggressively for Dragonfly actually hurts event efficiency even more. Traffic does not see quite as much improvement as Dragonfly, but in the best case, Traffic Dest sees a 15% improvement in event rate.

The original goal of this technique was to improve performance by having a positive effect on both event efficiency and total communication. Figure 4.27 shows the effect on event efficiency for each configuration. These results largely mirror the effects on event rate for PHOLD and Dragonfly. PHOLD Base actually decreases in event efficiency, PHOLD Work is not effected, and PHOLD Event and Combo both see an increase in event efficiency. For Dragonfly, we see an initial increase in event efficiency, but once too many events are held, it causes lag in event arrivals which negatively impacts event efficiency. For Traffic, we see far more drastic effects. The event efficiency of all model configurations sees a significant increase, with Traffic Src and Traffic Route roughly tripling their event efficiency. However, aggressively holding back so many events in these unbalanced model configurations starves some processors from work which is why we do not see speedups commensurate with the event efficiency increases. However, in the next chapter we will show that because of this, we can combine this technique with dynamic load balancing to achieve significant performance improvements.

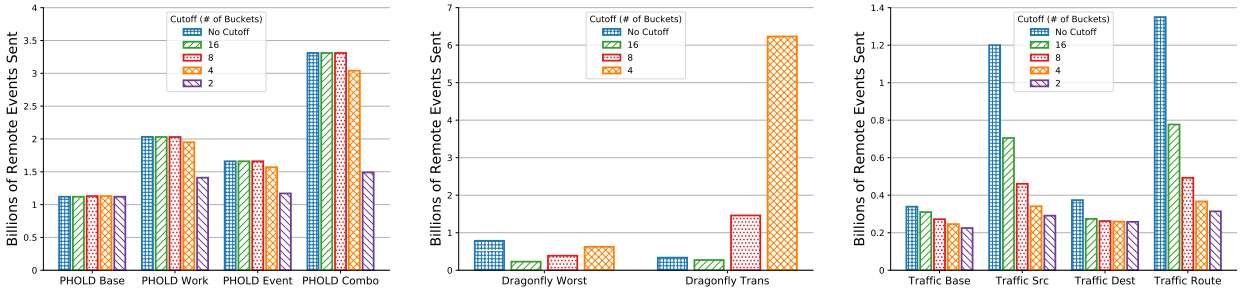

Figure 4.28: The number of events sent remotely for each model when throttling event sends based on their offset from the current GVT.

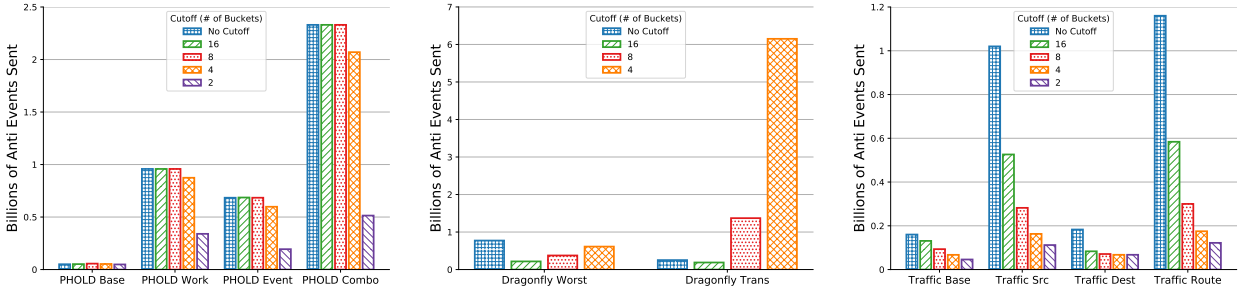

Figure 4.29: The number of anti-events sent for each model when throttling event sends based on their offset from the current GVT.

Finally, Figures 4.28 and 4.29 show the effects on the actual communication. Figure 4.28 shows plots the actual number of events sent remotely for each configuration. This of course, does not include events which were held back and canceled without being sent. As we the cutoff decreases we see the total amount of events sent remotely significantly decrease as well. For PHOLD configurations other than Base, we see communication cut by a factor of 30-55%. For Traffic and Dragonfly, the number of events sent is cut by up to 75% depending on the configuration. There is an even more pronounced effect when looking at the number of anti-events sent, where in the most extreme case anti-events are cut down by 90% for Traffic Route.

4.4.4 Summary

The Adaptive Bucketed GVT algorithm proposed here takes concepts from the Phase-Based GVT algorithm presented in Section 4.3 and improves upon them by making them virtual time aware. The algorithm is able to operate independently from the `Scheduler` without blocking event execution during the computation of the GVT. Event timestamps are used as extra information which allows the algorithm to adapt to the simulation execution

both in determining when to compute a GVT and what to include in the GVT computation. This results in a non-blocking GVT algorithm which scales effectively and requires less collective communication that the Phase-Based approach.

Furthermore, we showed an extension to the algorithm which can combat the primary drawback of non-blocking GVT algorithms as shown in this thesis: a low event efficiency. By taking advantage of the structure of the algorithm and the information it gathers, we can throttle event communication for events which are likely to be canceled and cause further rollbacks. In the cases studied here, this can have substantial benefits in terms of event efficiency and total communication. However, these benefits did not always translate into higher performance. As future work, we want to look at techniques to more effectively utilize the information present in the GVT to predict which events to hold and for how long. In doing so, we may be able to further increase the accuracy with which events are held to increase the benefits gained from this technique.

4.5 CONCLUSION

In this chapter we looked at three different algorithms and studied their effects on various simulation characteristics across all three of our models. We started by looking at a simple Blocking GVT algorithm, similar to the one used in ROSS and elsewhere. This algorithm required event execution to fully block for at least a portion of the computation. We showed how both the GVT frequency, and the trigger for computing a new GVT had significant impact on both the synchronization cost and event efficiency of the algorithm. However, these two quantities are inherently coupled together by the nature of the algorithm. This makes the performance extremely sensitive to tuning, and depending on the model, may require high synchronization costs to run effectively.

We then looked at two different algorithms which can operate alongside event execution, eliminating all need for event execution to block during GVT computation. The first of these algorithms, the Phase-Based GVT algorithm, is based off of work by Mattern and Perumalla. However, here we demonstrate an effective and scalable implementation which is able to take advantage of message-driven execution to adaptively overlap the GVT algorithm with event execution without requiring explicit communication scheduling by the simulator.

The last algorithm we look it is a new algorithm proposed in this thesis, and aims to improve upon certain drawbacks presented by the Phase-Based algorithm. This Adaptive Bucketed GVT algorithm is virtual time aware, and able to adapt to the particular simulation environment to maximize its effectiveness. This adaptivity means it requires far less tuning than either of the other algorithms presented in this chapter to run effectively. It shows

performance improvements over both the Blocking algorithm and Phase-Based algorithm. In particular, it is able to run in a non-blocking fashion with significantly less communication than the Phase-Based algorithm. Furthermore, it can utilize the data it collects to adaptively throttle event communication to improve event efficiency and cut down on the number of events and anti-events which get communicated between processors. As future work, it presents a solid foundation for more advanced introspection to more effectively control event efficiency and communication while still achieving low synchronization costs.

CHAPTER 5: DYNAMIC LOAD BALANCING

In the previous chapter, we explored a tradeoff between synchronization cost and event efficiency, and how the method of GVT computation plays a strong role in each of these factors. In many cases, enforcing more synchronization was able to increase the event efficiency. For the basic Blocking GVT algorithm, finding the best performance became a balancing act between synchronization cost and event efficiency. The two non-blocking algorithms almost entirely eliminated synchronization cost, but suffered a loss in event efficiency. Despite the loss in event efficiency, these two GVT algorithms led to the best performance for most model configurations studied. However, this also reveals a clear avenue for improvement. If the event efficiency could be improved independently of the GVT computation then we could achieve low synchronization cost and high event efficiency. The Adaptive Bucketed GVT algorithm demonstrated one way to achieve this: throttling event communication. In this chapter we show that dynamic load balancing can be used to further improve event efficiency, as well as improving overall performance by achieving a better balance of work.

One thing that was learned from the Blocking algorithm was that if some LPs are allowed to run further ahead of others, event efficiency will be lower. The fact that explicitly constraining LP progress using the leash-based trigger in Section 4.2 leads to higher event efficiency than the count-based trigger demonstrates this point. Another factor that affects LP progress which we have not yet touched upon is the layout of LPs to processors. Each processor schedules event execution independently, and does so by selecting the LP with the earliest event to execute that it knows about. As such, if a processor only has LPs that have events further in the future it will execute them and allow those LPs to get even further ahead. Conversely, if a processor has only LPs with a lot of events which are close to the current GVT it will take more work for this processor's LPs to catch up to others which may be further ahead. This is where dynamic load balancing comes in. By changing the mapping of LPs to processors during execution we may be able to beneficially affect the rate at which LPs are advancing, and create a more balanced execution. This should lead to a higher event efficiency, and therefore a higher event rate.

In Section 5.1 we first look at the load balancing framework in Charm++ and how it is utilized in Charades. Dynamic load balancing of LPs is largely enabled by the fact that they are entirely encapsulated in objects which the runtime system can migrate. We will discuss in more detail how this works and what changes are required by the simulator and in models which would like to benefit from dynamic load balancing. We then look at how the decisions on where to actually move the LPs are made. This comes in two pieces: load balancing

strategies and load balancing metrics. The strategies are the actual algorithms which decide how to remap LPs based on load characteristics of the current mapping. The metrics are how we quantify and attribute load to a given LP, which is of particular importance in this work due to the speculative nature of optimistic synchronization. Section 5.2 describes the strategies and metrics we will be exploring in this work.

In Section 5.3, we evaluate and analyze the different load balancing strategies for each model. In these initial experiments we focus on results using the Blocking GVT algorithm in an attempt to more easily isolate the effects of load balancing. We look at how each strategy affects event efficiency and balance of load across processors. As an unexpected side effect, we also look at how remapping load affects the synchronization cost of the Blocking algorithm. Experiments in this section are done using the default load metric in Charm++, which is CPU time taken by each objects execution.

Section 5.4 expands upon Section 5.3 by looking at a wide variety of load metrics. We show that just using CPU time can be ineffective in the face of speculative execution, and explore more PDES specific metrics which can be measured by the simulator. We also analyze how characteristics of each model configuration are captured by each metric. Using this we give some intuition on how to select metrics which best characterize imbalance in a given model.

Finally, in Section 5.5, we show how using dynamic load balancing in conjunction with non-blocking GVT algorithms can further improve their efficacy. The primary drawback of these algorithms was a low event efficiency. By adding dynamic load balancing, we can improve the event efficiency while still reaping the benefits of the non-blocking execution.

5.1 CHARADES LB FRAMEWORK

As discussed in Chapter 2, Charm++ offers robust support for dynamic load balancing. This support stems from the fact that the Charm++ runtime system manages the locations of parallel objects (called chares) for the user, as well as scheduling computation and communication for these objects. As such, there is a large body of research in dynamic load balancing that has been done on top of the Charm++ runtime system in many application domains and with varying goals [23, 25, 61, 26, 62, 27, 63, 24, 64, 65, 66, 67]. In this thesis, we build upon this research by exploring the effects of load balancing in optimistic PDES simulations where there is speculative execution. The speculative execution of events adds additional challenges to load balancing which we must address. First of all, effectively measuring the load of each object is not straightforward when considering the fact that work done by the object may not always contribute to the final simulation result. Secondly, changing the locations of objects may also change the total amount of work done

by the simulator by affecting the amount of speculative execution that requires rollbacks. To address these challenges, we test a variety of metrics for attributing load to objects which more effectively capture a useful notion of "load" in an optimistic simulation.

In Charades, load balancing is accomplished by migrating LPs during execution to achieve a more effective mapping of LPs to processors. In our design of Charades, described in Chapter 3, LPs are encapsulated within chares, which means that the Charm++ runtime system is able to manage their locations, computation, and communication. This also allows the runtime system to move LPs around during execution, as well as automatically measure load statistics for each LP. The combination of these two features allows the runtime to intelligently re-balance LPs to maintain an effective balance of load during execution of a simulation.

In order for an application to use dynamic load balancing it must do three things:

- Inform the runtime system how to serialize and deserialize objects
- Inform the runtime system when to perform load balancing
- Select one or more load balancing strategies and load metrics

Object serialization and load balancing timing is discussed in the following paragraphs. The specific strategies and metrics studied in this thesis are discussed in detail in the next section.

5.1.1 Object Serialization

In order to serialize and deserialize objects, the application uses the Charm++ Pack-and-UnPack (PUP) framework [68]. The PUP framework uses macros and operator overloading to allow applications to specify which parts of an object to migrate, and how to deal with more complex structures involving pointers. Each chare type has a virtual `pup` function defined in its base class, which users can override to instruct the runtime on how to migrate that chare. Migration occurs in three steps: sizing, packing, and unpacking. During each step, the same `pup` function is called, and passed a reference to a `PUP` object which behaves differently based on which step is being executed. During the sizing step, the `PUP` object determines the total size of the data being migrated, and allocates an appropriately sized buffer. During packing, the data to be migrated is copied to the buffer. The buffer can then be sent to the chares new location. During unpacking the data is copied from the buffer to appropriate fields of the newly allocated chare. Any fields not included in the `pup` function will not be migrated. An example pup function is shown in Listing 5.1. The `|` operator is overloaded and used to tell the `PUP` object which object members should be migrated.

Listing 5.1: Example PUP function

```
1  class ExampleChare {
2  public:
3    int member1;
4    SimpleObject member2;
5    ComplexObject* member3;
6    OtherObject member4; // Some transient state that doesn't need to be migrated
7
8    void pup(PUP& p) {
9      p | member1; // Tell p to size, pack or unpack member1 depending on state of p
10     p | member2; // Same for member2
11
12     // Since member3 is heap allocated, we need to check if we are unpacking so
13     // that we can allocate space for it before unpacking it
14     if (p.isUnpacking()) {
15       member3 = new MemberObject();
16     }
17     p | member3; // Size, pack, or unpack member3
18   }
19 };
```

Primitive types and flat objects can be directly piped through the `PUP` object as shown on lines 9 and 10. If `SimpleObject` contains its own pup function then it will be called to recursively migrate `member2`. Otherwise, it will just treat `member2` as a flat chunk of bytes the same size as `SimpleObject`. For pointers to data, an extra step needs to be taken to ensure that space is correctly allocated during the unpacking step of migration. This is shown on lines 14 to 16 before piping `member3` through the `PUP` object on line 17. In this example, `member4` does not need to be migrated and is therefore not included in the `pup` function. This reduces the amount of data sent across the network during migration.

In Charades, LPs are implemented as chares, which allows the runtime to automatically track their load and migrate them during load balancing. The LP chares have a `pup` function to correctly serialize all the required simulator information for each LP, including things like pending and passed event lists, causality information, cancellation information, and current virtual time. By default, it migrates the model specific state as if it were a contiguous block of memory; however complex models can provide their own PUP function handle for LPs that have state that requires more careful serialization.

5.1.2 Load Balancing Synchronization

In terms of when to perform load balancing, the application must let the runtime system know when it is at a point at which it is safe to migrate chares. For complicated applications,

migrating a chare at an arbitrary time may cause issues, especially if chares are using shared data structures. Because of this, the runtime system provides a few methods for informing it when load balancing can occur. For this work we will focus on the most common of these methods, `AtSync()`. In this mode, `AtSync()` is a function provided by the runtime that must be called on every chare before migration can occur. Once every chare calls `AtSync()`, the runtime knows it is safe to start load balancing. At this point, the runtime system calls the specified load balancing strategy which collects load data from each object and makes a decision on how to redistribute the objects. Once the decision has been made, migrations are performed, and optionally a `ResumeFromSync()` method is called informing chares that load balancing has been completed. Based on application needs, chares can block until `ResumeFromSync()` is called, or continue execution immediately after calling `AtSync()`.

For Charades, we chose to synchronize for load balancing immediately after the completion of certain GVT computations. This decision was made for a couple of reasons. First, depending on the GVT algorithm chosen, the GVT computation may already be an existing synchronization point within a simulation, which provides a natural place to do load balancing. How frequently, and how many times, to perform load balancing is a runtime parameter. Secondly, after the GVT computation completes, fossil collection occurs which frees up unneeded memory within LPs. Calling `AtSync()` right after fossil collection will minimize the amount of data that has to be migrated between processors. In the case of blocking GVTs, the simulation resumes after load balancing has completed. For non-blocking GVT algorithms, we would like to avoid blocking the simulation for load balancing as well. In these cases, we have also experimented with allowing load balancing to be overlapped with the simulation itself, much like the GVT algorithm. In this case, we simply ignore the `ResumeFromSync()` message from the runtime and continue the simulation immediately after calling `AtSync()`. Exactly which GVTs to perform load balancing after is configured via runtime parameters. In this work, we manually tune these parameters experimentally, however there has also been work done within the Charm++ runtime system to automatically select load balancing frequency based on application performance characteristics [24].

5.2 STRATEGIES AND METRICS

This section describes a number of strategies and metrics used during the load balancing experiments done in the next sections. Load balancing strategies are the actual algorithms which decide how to remap objects, and come in both centralized and distributed forms. The strategies used in a particular application can be chosen at runtime or compile time. Research has also been done to allow machine learning algorithms to select an appropriate strategy

based on application characteristics, however we will not be using that in this work [24]. As input, each strategy takes a the current mapping of objects to processors, where each object has some load. Some strategies may also take communication into account by taking a communication graph as an additional input. The strategy then uses these inputs to determine how to remap the objects to processors in order to get a better balance of load across processors. The primary difference between centralized strategies and distributed strategies is whether or not the inputs are gathered to a single processor before making the decision.

Metrics are the measure used to attribute load to a particular object. By default, the load of an object is automatically measured by the runtime system as the amount of CPU time spent executing methods of that particular object. For many traditional HPC applications this works well, as it allows load balancing strategies to balance the CPU resource needs across processors. For optimistic PDES simulations, speculative execution can mean that CPU time is no longer an accurate measure of meaningful work done by a given object. However, the runtime system also allows us to specify application specific metrics. Application specific metrics allow the simulator to attribute load to each LP in a way which is more meaningful to a PDES simulator. These metrics can capture different types of imbalance in the system and can be used to achieve different end results based on the needs of the model and simulator configuration being used.

5.2.1 Strategies

Load balancing strategies we will be exploring in this chapter can be broken into two different types: centralized and distributed. Centralized strategies gather all load data to a single processor before making the load balancing decision. Distributed strategies make the load balancing decision in a distributed fashion, and include explicit communication of load data.

Centralized Strategies

Centralized load balancing strategies use a reduction tree to collect all load information on a single processor, which then uses this load information to make the complete load balancing decision. This decision is then broadcast to the remaining processors, and the necessary migrations are performed. With PDES, which frequently has a large number of fine-grained objects and messages, centralized strategies can require a large amount of data to be sent to a single process and incur a high overhead for decision making time. However,

Algorithm 5.1 GreedyLB

```
1: procedure GreedyLB::balance(ObjMap objs)
2:     ProcHeap procs;          ▷ Heap of empty processors, lightest loaded at the top
3:     objs.sort();                          ▷ Sort objects so largest are at the front
4:     int i = 0;
5:     while i < objs.size() do      ▷ Iterate through objects from heaviest to lightest
6:         objs[i].new_proc = procs.top();
7:         objs[i].new_proc.load += objs[i].load;
8:         procs.heapify();
9:         i++;
10:    end while
11: end procedure
```

due to the fact that the process making decision has complete information about the whole system, centralized strategies can provide a very effective load redistribution that can at least be used as a baseline to compare lower overhead distributed strategies against. In this work we primarily use a greedy centralized algorithm and an extension of the basic greedy algorithm which attempts to limit migrations.

GreedyLB is a centralized strategy which employs a basic greedy algorithm to assign LPs to processors. As shown in Algorithm 5.1, it sorts objects by load then iteratively assigns the heaviest unassigned LP to the least loaded processor until all objects have been assigned. Because the algorithm does a full reassignment without taking an objects current location into account, it often results in almost every LP being migrated.

GreedyRefineLB is a variation of GreedyLB which attempts to limit the number of migrations required while still providing good load balance. The algorithm is shown in Algorithm 5.2. It attempts to approximate GreedyLB by using two different parameters to limit migrations: `A` and `B`. `A` is the parameter which control how much we allow the max loaded processor to vary. In line 2 we multiply the estimated max load by `A` to get `M`. Max load can be estimate either by running GreedyLB or by using a heuristic based on average load. `M` is then used on line 9 to see if leaving an object where it is would cause a processors load to exceed the estimated max by more than is allowed. `B` is the second parameter, and controls how much we allow load to vary between two processors for individual migrations. It is used on line 9 to check if an the load of an objects current processor is similar enough to the ideal new processor. If leaving an object on its current processor would not cause that processor to exceed a target maximum, and would also result in a similar load as the ideal migration target, than the object stays on its current processor (line 10). On inspection, the structure of the algorithm is near identical to GreedyLB, other than the additional check to

Algorithm 5.2 GreedyRefineLB

```
1: procedure GreedyRefineLB::balance(ObjMap objs,A,B)
2:     M = EstimateMax() * A;          ▷ Attempt to limit max load within factor of A
3:     ProcHeap procs;          ▷ Heap of empty processors, lightest loaded at the top
4:     objs.sort();                          ▷ Sort objects so largest are at the front
5:     int i = 0;
6:     while i < objs.size() do      ▷ Iterate through objects from heaviest to lightest
7:         min = procs.top();
8:         prev = objs[i].prev_proc;
9:         if prev.load + obj.load <= M && prev.load <= min.load * B then
10:            objs[i].new_proc = prev;                    ▷ Keep object where it is
11:        else
12:            objs[i].new_proc = min;
13:        end if
14:        objs[i].new_proc.load += objs[i].load;
15:        procs.heapify();
16:        i++;
17:    end while
18: end procedure
```

potentially leave an object on its current processor. It can also be seen that if `A=0` or `B < 0` that the algorithm will behave identical to greedy.

Distributed Strategies

Unlike centralized strategies, distributed strategies do not require all load information to be sent to a single location. Furthermore, every processor takes part in the decision process, and the coordination amongst processors and how data moves between them is a core part of the strategy itself. At the start of load balancing, each processor gathers load statistics for its own local objects. From there, the way the data is communicated and the decision is made is entirely up to the strategy.

GrapevineLB is a strategy that utilizes probabilistic migration and a gossip protocol as described in [69] 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 know about some of the under loaded processors in the system and asynchronously attempt to shift some of their load to the them in a probabilistic fashion. Each overloaded processor takes its smallest object and randomly selects an under loaded processor that will not become overloaded if the selected object is

reassigned to it. It repeats this process until it has sent enough objects that it is within some target threshold of the average load, or until it has no objects that fit on any of the under loaded processors it knows about. It then sends information about potential transfers to each processor it is sending objects to, and waits for a positive or negative acknowledgment, because it is possible that other processors also added load to the target processor and it can no longer accept incoming objects. In initial experiments, this did little to affect the performance of our simulations due to very few objects successfully migrating due to collisions in the probabilistic migration attempts. For this work, we have modified the GrapevineLB 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 also means that the most heavily overloaded processors get multiple attempts to transfer load, which prioritizes reducing the ratio of max to average load more aggressively than the original implementation.

5.2.2 Load Metrics

Each strategy described above runs based on the "load" of the LPs being simulated. By default, load is just the CPU load of each LP; however, the Charm++ runtime allows applications to specify their own measure of load for each object which is something Charades can exploit to more precisely define the load of each LP. In PDES, specifically optimistic PDES, the CPU load of an LP might not be the best way to attribute load to that object. This is due to the fact that the CPU load also includes time spent rolling back events which represents work that is not a part of the final simulation result. Because of this, we've chosen to evaluate a number of more PDES specific metrics with each of the strategies above. These metrics can be largely split into two different categories: past-looking metrics, and future-looking metrics. Past looking metrics specifically utilize information about events that an LP has already executed as a predictor of what the LP will probably be doing in the near future. Future-looking metrics attempt to more directly predict the future load of an LP by using information about events and work that the LP has yet to perform but will likely perform.

Past-Looking Metrics

Committed Events: This metric uses the number of committed events since the last load balancing as the weight of an LP. It attempts to capture the notion of "useful work" executed by an LP by ignoring time spent falsely executing events, rolling back events, and other overheads associated with event management. It does not take into account time spent actually executing the events in question, and therefore probably works best when each event takes roughly the same amount of time.

Past Events: This metric is similar to the first metric, but adds the number of processed events to the weight of an LP. It attempts to capture all work spent executing events, while ignoring overheads for event management and rollbacks. It may be less effective for cases where the LP has a very low event efficiency and therefore rolls back a large portion of processed events, however if this rollbacks are for some reason inherent to the model and the execution pattern of the LP then it is possible that this work will be performed regardless of LP placement and therefore should still be attributed to an LPs weight.

Executed Events: Even more inclusive than Committed Events and Past events, Executed Events is simply a count of all events executed since the most recent load balancing. This includes events which are executed but eventually rolled back. It basically attempts to capture all forward work of the simulator while ignoring rollback and queuing costs.

Current Time: This metric attempts to balance based on the rate of progress of each LPs. The actual weight attributed to an LP is normalized to $\frac{t}{t_{end}}$ where t is the current virtual time of the LP, and t_{end} is the end time of the simulation. It attempts to capture the rate of progress of each LP so that the load balancer can balance the rate of progress of each processor. For all of the timestamp based metrics, we have also tried inverted versions of the metric $(1 - \frac{t}{t_{end}})$ and found that they largely perform the same.

Latest Time: Similar to Current Time, this metric attempts to capture rate of progress of an LP. However, here we look at the latest time the LP has ever reached, even if it has since been rolled back. This more directly captures LPs which may be suffering from low event efficiencies.

Future-Looking Metrics

Pending Events: This metric is fairly straightforward, in that it looks at the size of each LPs queue of pending events and uses that as the weight of the LP. LPs with more pending events are assumed to have more work to do and should therefore receive more resources. This metric also therefore relies somewhat on a high event efficiency otherwise many events

in pending queues may be rolled back or canceled before they are even executed.

Weighted Pending Events: Weighted pending events attempts to improve the previous metric by weighting events in each LPs pending queue by how soon they will be executed, the idea being that earlier events are closer to the current GVT and less likely to be rolled back or canceled. Furthermore, they are events that will be executed in the more immediate future whereas events that are further away in virtual time may not be executed for a while as more events are generated via the execution of the sooner events. The weight is computed using the events timestamp in the same way it was used for the Current Time and Latest Time metrics.

Active Events: Active events are simply all events which may still result in some further action being taken in the future. This is the sum of all events in both the pending queue (which may be executed), and in the processed queue (which may be committed or rolled back). It is a very broad metric which simply attempts to include any upcoming work in the load balancing decision.

Next Time: Almost identical to the past-looking metric, Current Time, Next Time attempts to capture LP progress through virtual time by looking at the next event each LP will execute. This can also be thought of as a much simpler version of Weighted Pending Events. Unlike Current Time, if an LP has no pending events, or ones very far in the future, it will treat that LP as very light due to the fact that it may not have any work to execute for a while.

5.3 STRATEGY COMPARISON

In this section we evaluate each strategy discussed in Section 5.2 on the unbalanced model configurations for PHOLD and Traffic. As with the previous chapter, the ultimate goal of adding dynamic load balancing to Charades is, of course, to achieve high event rates on a variety of models. However, we will also be examining how these event rates are achieved, specifically looking at the effects of load balancing on synchronization costs and event efficiency. Like with the GVT work, these two elements are important factors that effect the overall event rate of a simulation, however in this section we see less of a direct tradeoff in the two quantities. For now, we will limit our analysis to configurations using the Blocking GVT algorithm. This allows us to look at the effects of load balancing in a more isolated manner, due to the fact that we can easily snapshot simulation state before and after load balancing occurs. We also start by only using the default load metric of CPU time. This is the default method for attributing load in Charm++ and is also common in other HPC applications and frameworks. In the next section, we will see how the results

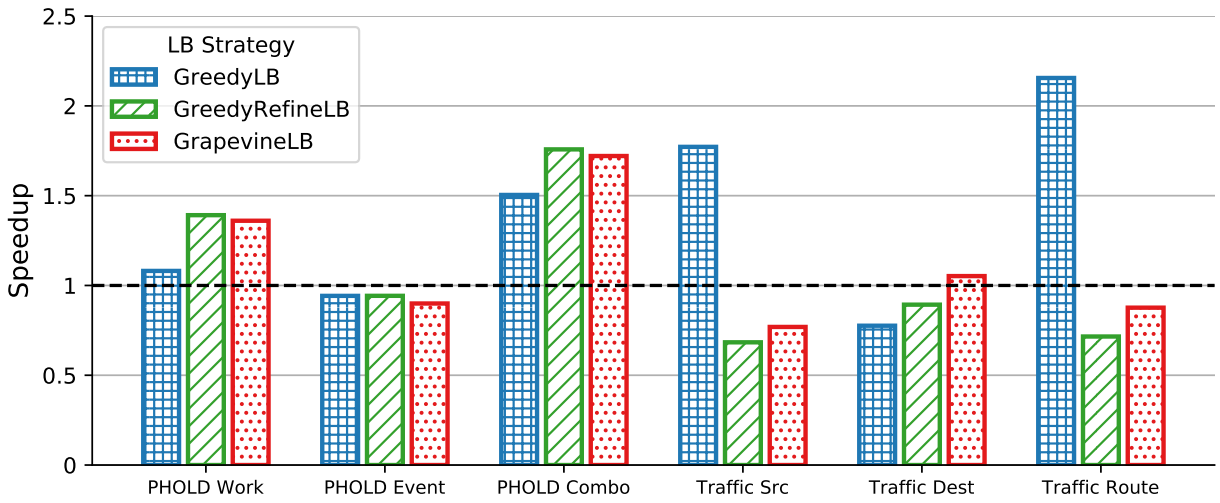

Figure 5.1: Speedup of each unbalanced model configuration under each load balancing strategy. Each model configuration uses the count-based GVT trigger.

change under various metrics and demonstrate some possible drawbacks of using CPU time as load in optimistic PDES.

It is also important to note that since we start our evaluations by focusing on the Blocking GVT algorithm, we must also consider the effects of the GVT trigger. The GVT trigger can have a significant impact on how load balancing performs because it indirectly dictates how much work a given LP will do during each GVT interval. Since load balancing occurs at certain GVT boundaries, this means that the trigger dictates which work will have been completed when load balancing begins. This will become even more impactful later on when we discuss load balancing metrics. For this section, the main distinction to note is that the count-based trigger keeps the number of events executed on each processor the same. For models where each event takes roughly the same amount of time, such as PHOLD Event or any of the Traffic configurations, this means that each processor will take roughly the same amount of CPU time executing events. The differences in load will primarily come from the extra amount of time some processors spend performing rollbacks. For the leash-based trigger, this constraint on number of events executed is no longer there, and processors may see much larger differences in compute time required to reach the next GVT.

5.3.1 Performance

To start our analysis, we first look at the overall effects of load balancing on each model configuration under each GVT trigger. Unless otherwise specified, runs in this section were performed on 64 nodes (2,048 processes) of Blue Waters. Figure 5.1 shows how load balancing

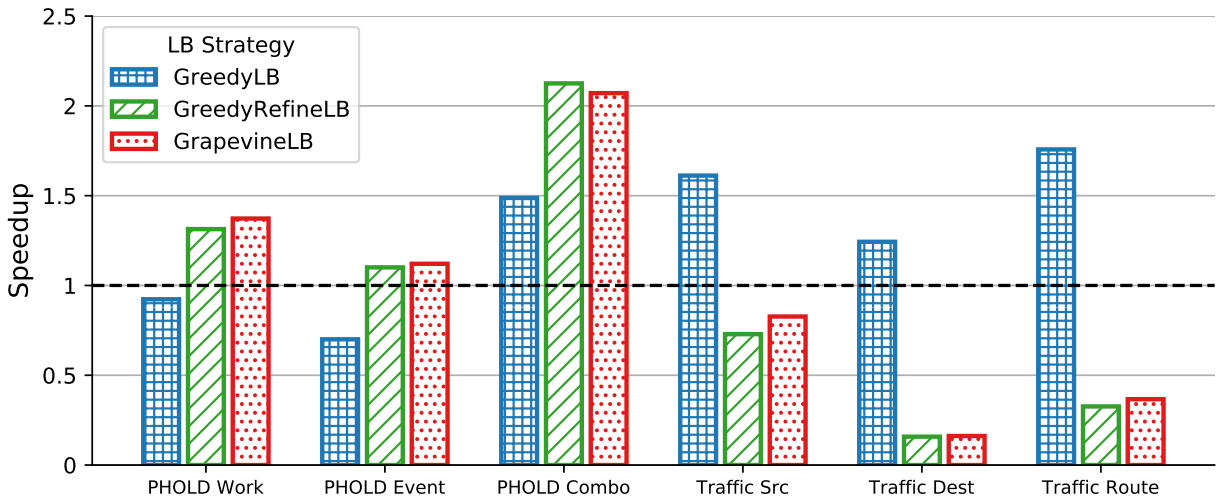

Figure 5.2: Speedup of each unbalanced model configuration under each load balancing strategy. Each model configuration uses the leash-based GVT trigger.

affects each model under the count-based GVT trigger by showing speedup with each strategy compared to runs with no load balancing. Similarly, Figure 5.2 shows the speedups for model configurations under the leash-based GVT trigger. All speedups are computed by taking the event rate for the entire run and dividing by the event rate for the run without load balancing. This means that the speedup includes execution before load balancing, as well as the time taken by load balancing itself. In almost every configuration there is at least one load balancing strategy that leads to speedup. In particular, for the leash-based metric, we see speedups for every single configuration. In all configurations, the leash-based trigger outperformed the count-based trigger, and that holds true when a good load balancing strategy is used as well. We also see a stark contrast in which strategies perform well for each model. PHOLD generally gets the best performance from GreedyRefineLB and GrapevineLB, whereas Traffic does best with the full re-balance provided by GreedyLB.

Figures 5.3 and 5.4 give us insight into what the load balancing framework sees for each PHOLD model configuration in terms of load per processor. This highlights the effects of using the Blocking algorithm with the different triggers while load balancing. For the PHOLD Event configuration, where each event takes roughly the same amount of time to process, we see that when using the count trigger there is very little variation in load across processors. This comes despite the fact that we have created an imbalance in the distribution of events. This makes it difficult for the load balancers to effectively redistribute load as demonstrated in Figure 5.1. The other two PHOLD configurations have imbalances in the amount of time taken to execute events, which results in approximately a 2× ratio of max

to average load. Because of this, load balancing has a much more positive impact on these configurations. Particularly, GreedyRefineLB and GrapevineLB are both able to effectively balance load because they depend on the max/average ratio in their decision making. The leash trigger does not constrain the number of events executed on each processor during each GVT interval. This results in a more distinct load variance across processors where there is imbalance in the number of events. This allows load balancing strategies more to effectively determine how to redistribute load. Figure 5.2 shows larger speedups than we saw with the count trigger for both PHOLD Event and PHOLD Combo, even considering the fact that they already had a higher event rate with the leash trigger.

The imbalance in the Traffic model configurations is entirely based off of event distribution, and Figures 5.5 and 5.6 back this up. Just like with PHOLD Event, the count metric hides imbalance by constraining the amount of work done per GVT interval. Any variation comes from rollback work. Again, Figures 5.1 and 5.2 show that Traffic Src and Dest get better speedup when using the leash metric. Traffic Route sees comparable speedup, and ultimately gets better performance when using the leash metric plus load balancing. However, the other two strategies, which try to limit migrations, perform poorly for the Traffic model across the board. In fact, the reason GreedyLB does well is that it essentially redistributes all objects, which naturally breaks up the concentrated areas of LPs which are unbalanced. This is evidenced by the fact that GreedyLB even does well when there is little discernible imbalance across processors when using the count metric. GreedyRefineLB and GrapevineLB have a very small effect on performance because their migrations are explicitly based on differences in max and average load ratio of processors, of which there is little for the count based metric. GreedyLB does not look at aggregate measures based on processors and simply examines each object one at a time and decides where to map it. For the leash metric, an imbalance across processors exists, but GreedyRefineLB and GrapevineLB still do poorly. This points to the fact that CPU load might not accurately capture the imbalance in a way that is useful to the load balancing strategies. In the next section we will look at other metrics which do a better job.

To determine where the speedups come from, we again look at both event efficiency and synchronization cost. The effects on event efficiency come from the fact that remapping LPs will change the speed at which a processor progresses through virtual time by changing the distribution of events. The effects on synchronization cost come from the fact that unbalanced workloads can lead to some processors lagging behind. By lagging behind, these slow processors cause other processors which have already reached the GVT computation to wait longer. The effects on synchronization cost are much more in line with the traditional benefits of load balancing in other applications, whereas the effects on event efficiency have

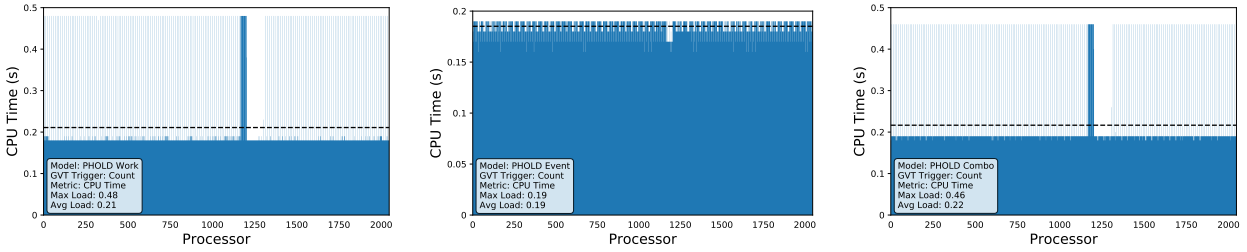

Figure 5.3: CPU load distribution across processors for each PHOLD configuration using the count-based GVT trigger.

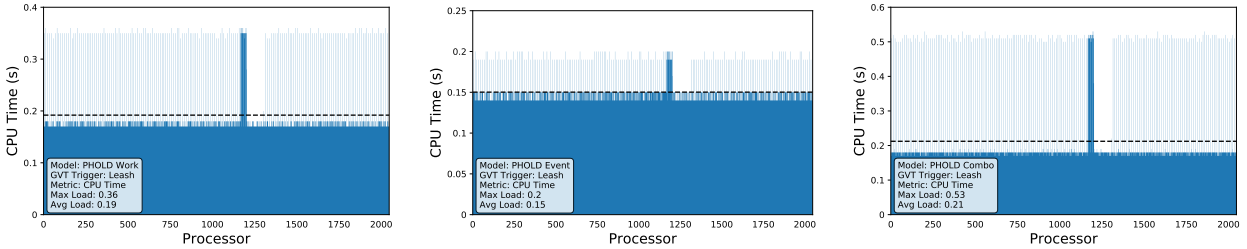

Figure 5.4: CPU load distribution across processors for each PHOLD configuration using the leash-based GVT trigger.

to do with how load balancing interacts with event scheduling and speculative execution. As such, these effects are somewhat unique to PDES.

5.3.2 Event Efficiency

Tables 5.1 and 5.2 show how load balancing effects event efficiency for each model under the count trigger and leash trigger respectively. For the count trigger, we see that every model but Traffic Dest has at least one load balancing strategy which increases event efficiency. Efficiency gains are much less drastic when using the leash trigger, largely due to the fact that event efficiency is already fairly high to begin with. However, for PHOLD will still see event efficiency increase when load balancing is used. These effects are particularly strong when considering configurations which have differently weighted events. Both PHOLD Work and PHOLD Combo contain some LPs which take significantly longer to execute events than others. In the initial mapping, the "heavy" LPs which take longer to execute events are clustered in a contiguous block. As we saw in the previous chapter, the discrepancy in the amount of real time it took processors to complete a GVT interval (whether it was determined by number of events or virtual time) caused a low event efficiency. This is due to fast moving processors getting ahead of slow moving processors, and then needing to rollback. In this case, CPU time is an effective metric to pick out these heavier LPs and

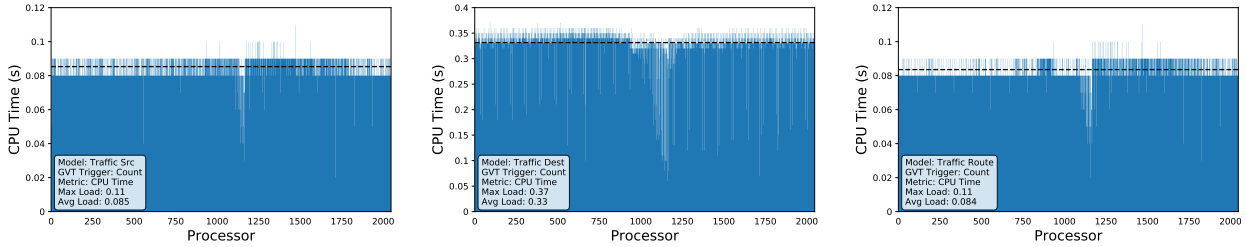

Figure 5.5: CPU load distribution across processors for each Traffic configuration using the count-based GVT trigger.

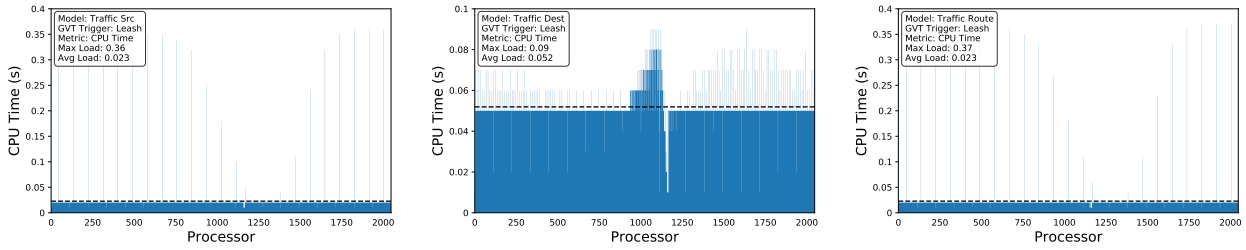

Figure 5.6: CPU load distribution across processors for each Traffic configuration using the leash-based GVT trigger.

load balancing is able to distribute them effectively. This evens out the speed at which each processor moves through the simulation, which helps prevent certain processors from getting significantly far ahead.

For models with an imbalance in event distribution, the resulting event efficiency increase comes from the fact that load balancing ends up creating a more even distribution of events. LPs which have more events in a given window of time will require more CPU time to execute these events than LPs which have few events to execute. In this sense, CPU time essentially becomes an estimate for the number of events each LP has in a certain time interval. By load balancing based on this estimate, each load balancing strategy is essentially trying to even out the number of events on each processor. This results in each processor moving more uniformly through virtual time and therefore creating less opportunity for costly rollbacks. However, it is again important to note that for Traffic, GreedyRefineLB and GrapevineLB struggle to improve event efficiency. Where GreedyLB remaps all objects based on their own weight, the aggregate per-processor measures that GreedyRefineLB and GrapevineLB use to determine when to move load may fail to see LPs which have a disproportionate number of events if they are hidden amongst lightly weighted LPs. With such low event efficiencies in the base model, the difference in CPU load being witnessed is also more likely to come from the heavy amount of rollbacks and cancellations required. So for Traffic, balancing based on CPU may not actually be achieving an even distribution of events to processors.

	PHOLD			Traffic		
LB Strategy	Work	Event	Combo	Src	Dest	Route
No LB	75%	55%	54%	12%	51%	11%
GreedyLB	95%	88%	95%	14%	19%	15%
GreedyRefineLB	88%	89%	92%	8%	30%	9%
GrapevineLB	88%	55%	89%	9%	36%	11%

Table 5.1: Event efficiency with each LB strategy using the count-based GVT trigger.

	PHOLD			Traffic		
LB Strategy	Work	Event	Combo	Src	Dest	Route
No LB	76%	84%	93%	97%	96%	97%
GreedyLB	96%	96%	96%	71%	85%	71%
GreedyRefineLB	95%	96%	96%	91%	45%	83%
GrapevineLB	95%	95%	94%	96%	51%	84%

Table 5.2: Event efficiency with each LB strategy using the leash-based GVT trigger.

5.3.3 Synchronization Cost

In the previous chapter, when we looked at synchronization cost of the GVT algorithm, we focused primarily on the frequency of the GVT computation as well as how long the actual computation took to complete. Load balancing also has an effect on the synchronization cost of the GVT algorithm. However, in load balancing, changes to the synchronization cost come in the form of better balance of work across processors. In the results shown in this chapter, the load balancing has no effect on the GVT interval, nor the cost of the GVT computation itself. Instead, load balancing changes the amount of work required by each processor to reach the GVT computation. If a better balance of work is achieved, then processors will reach the GVT computation at a similar time. This will result in less idle time while fast processors wait for slower processors to reach the GVT. Due to the effects of load balancing on event efficiency, we may also see changes in the number of GVT computations as well.

Figure 5.7 shows the number of GVT computations performed by the simulator in each of our model configurations for the count-based GVT trigger. In each case, at least one load balancing almost always reduces the number of GVT computations performed. This is a side effect of load balancing increasing event efficiency as shown in Table 5.1. Higher event efficiency means fewer total events executed. For the count trigger, the number of

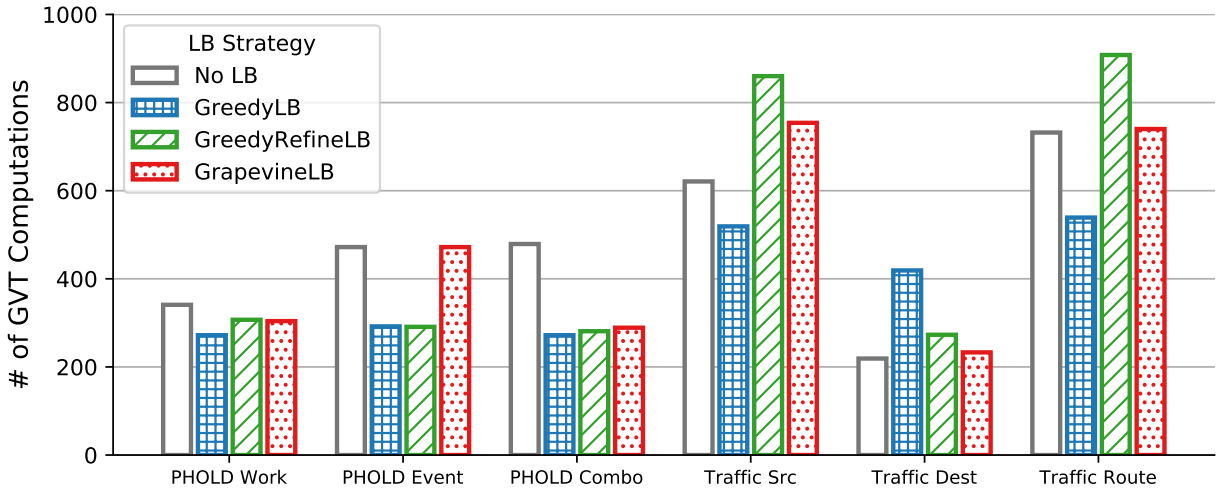

Figure 5.7: The number of GVT computations performed for each model configuration with each load balancing strategy. The GVT is configured using the count-based GVT trigger.

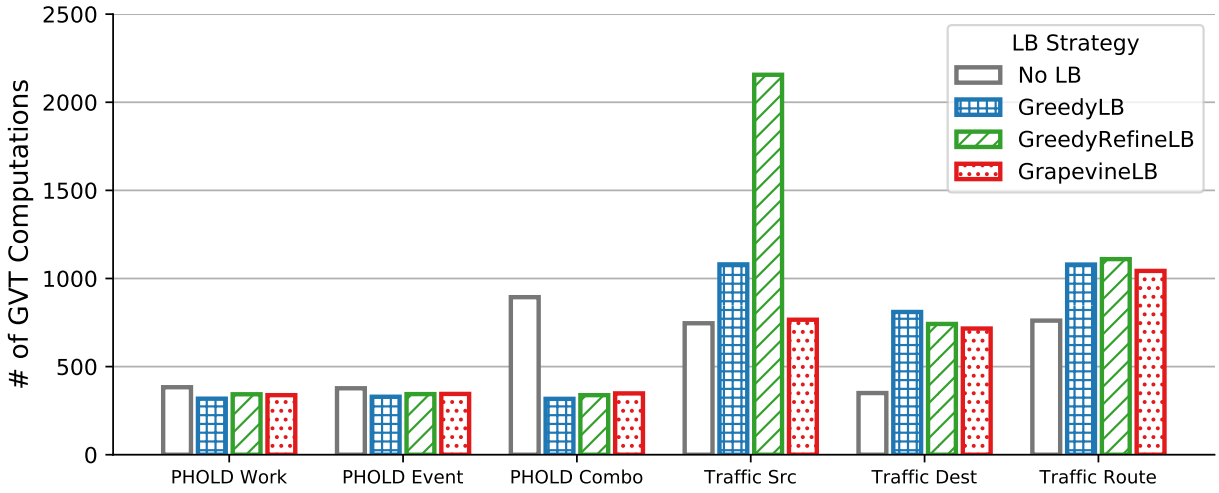

Figure 5.8: The number of GVT computations performed for each model configuration with each load balancing strategy. The GVT is configured using the leash-based GVT trigger.

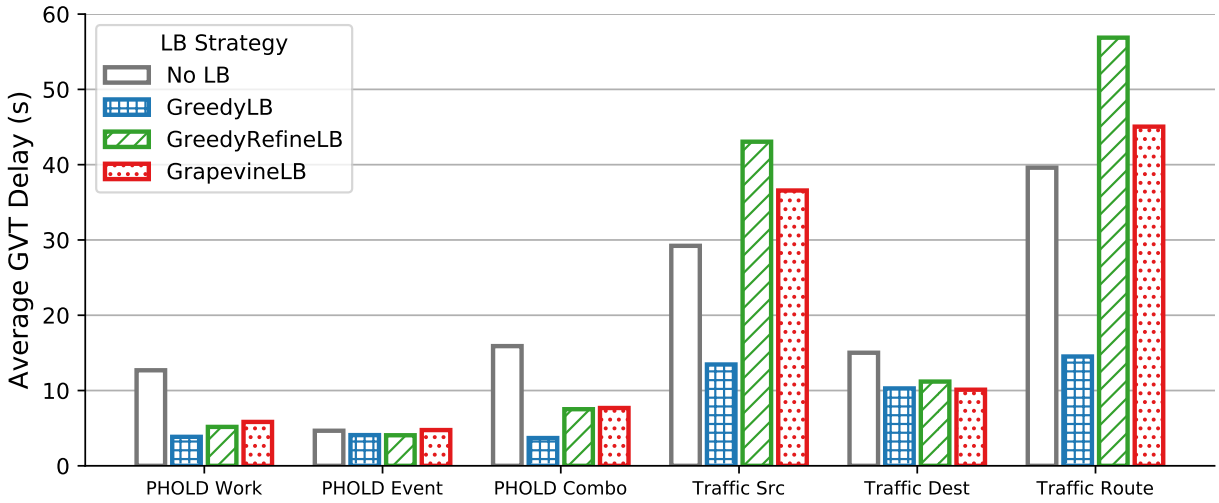

Figure 5.9: The total amount of time event execution is delayed by the GVT computation, averaged across processors for the count-based GVT trigger.

GVT computations is a function of the total number of event executions. Therefore, higher event efficiency results in fewer GVT computations. Figure 5.8 shows the same data, but when using the leash-based GVT trigger instead. We see a much less drastic effect. We also see an increase in GVT computations for the Traffic models. These difference between the count and leash based triggers occur primarily for two reasons. First, as seen in Table 5.2, the effect of load balancing on event efficiency is smaller for models using the leash-based trigger. Secondly, for the leash based trigger, the number of GVT computations is a function of total virtual time and leash size, not the number of events executed. However, events that arrive on a processor that has already triggered the GVT can still cause rollbacks and result in a lower GVT for that computation. In cases with very low event efficiency, this can still have a significant impact on the number of GVT computations.

More important than the number of GVT computations is the effect that load balancing has on the actual time blocking event execution. Figures 5.9 and 5.10 show the average amount of time each processor blocked event execution for the GVT computation under the count and leash-based triggers respectively. This was computed the same way as in Figure 4.4 from Chapter 4. We see that in the majority of configurations for both trigger types, the amount of time blocking on the GVT is decreased when load balancing is used. In both cases, this is due to a better distribution of work and events. Regardless of the trigger used, a balanced distribution of work and events will cause processors to reach the triggers at a more uniform time. Since the GVT algorithm itself was not changed the decrease in blocking time is entirely due to when processors reach the GVT computation. This is

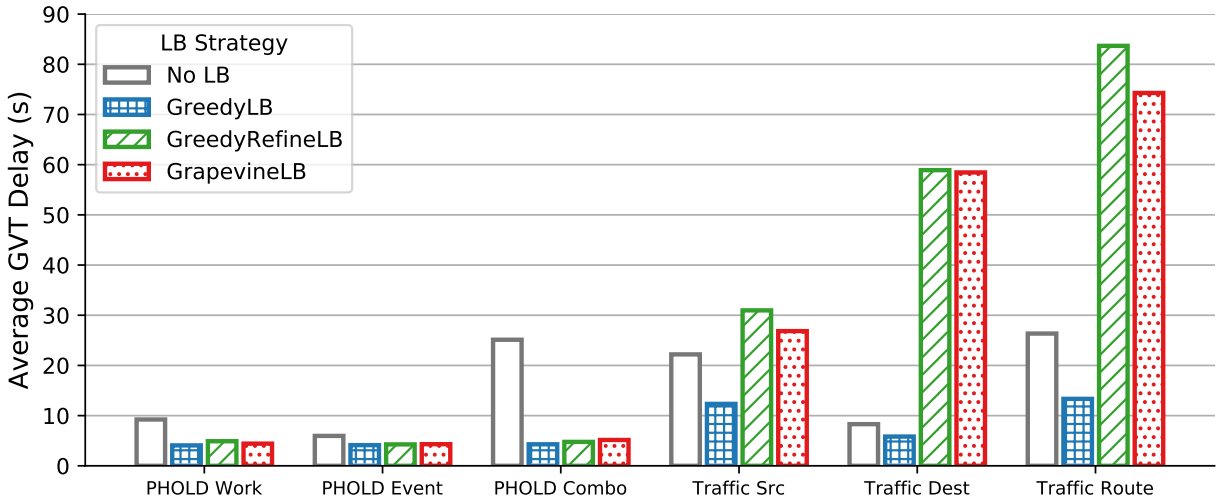

Figure 5.10: The total amount of time event execution is delayed by the GVT computation, averaged across processors for the leash-based GVT trigger.

reinforced by looking at Figures 5.11 and 5.12 which show the time spent blocking per GVT computation. These figures show that, on average, each GVT computation blocks for less time when a better balance is achieved, regardless of how the balancing may affect the total number of GVT computations done.

In order to get a more detailed view on the actual balance of synchronization cost across processors, we want to see more than just the average cost of each GVT computation. It is good to see that the average decreases, however it says nothing about the balance across the whole system. Figures 5.13 and 5.14 capture the distribution of GVT delays across processors by plotting a bar showing the range of GVT blocking times for each configuration. The top of each bar represents the maximum processor which blocked for the longest. The bottom of each is the processor which blocked for the least amount of time. The line in the middle of each bar represents the average across processors, which is the same as which was plotted in Figures 5.9 and 5.10. As the figures show, for most model configurations the range of time that processors spend blocking is decreased when using the correct load balancing strategy. More importantly, in terms of measuring load balancing effectiveness, the maximums are also decreased in most cases. Without load balancing, overloaded processors cause other processors to wait at the GVT trigger idle while the overloaded processors catch up. By balancing work, the overloaded processors have workloads closer to the average, resulting in less idle time for faster processors waiting for everyone to reach the GVT computation.

Another important factor to take note of is how the particular GVT trigger affects synchronization cost under load balancing. In general the virtual time leash-based trigger shows

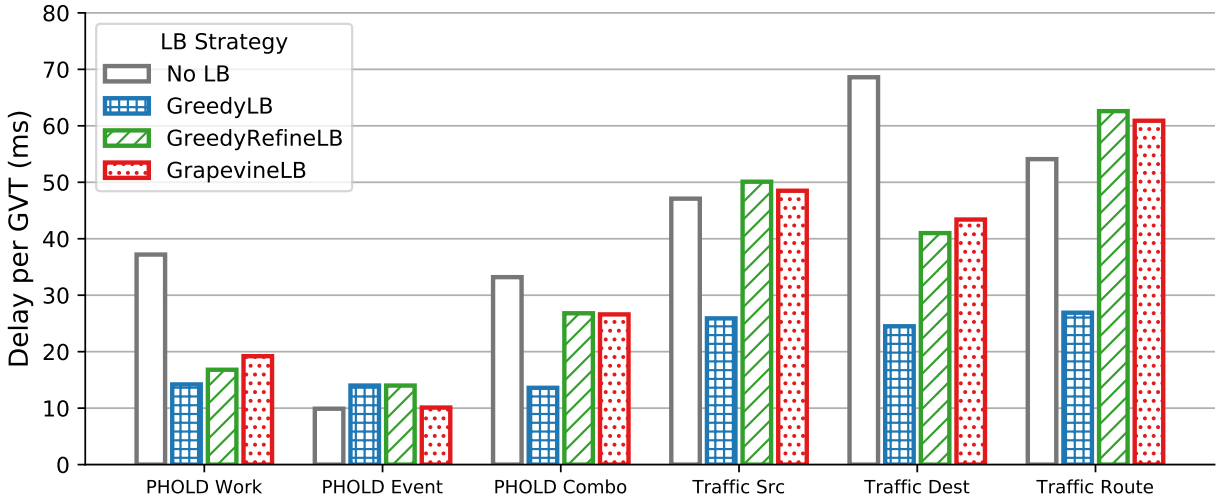

Figure 5.11: The average amount of time spent blocking per GVT computation for the count-based GVT trigger.

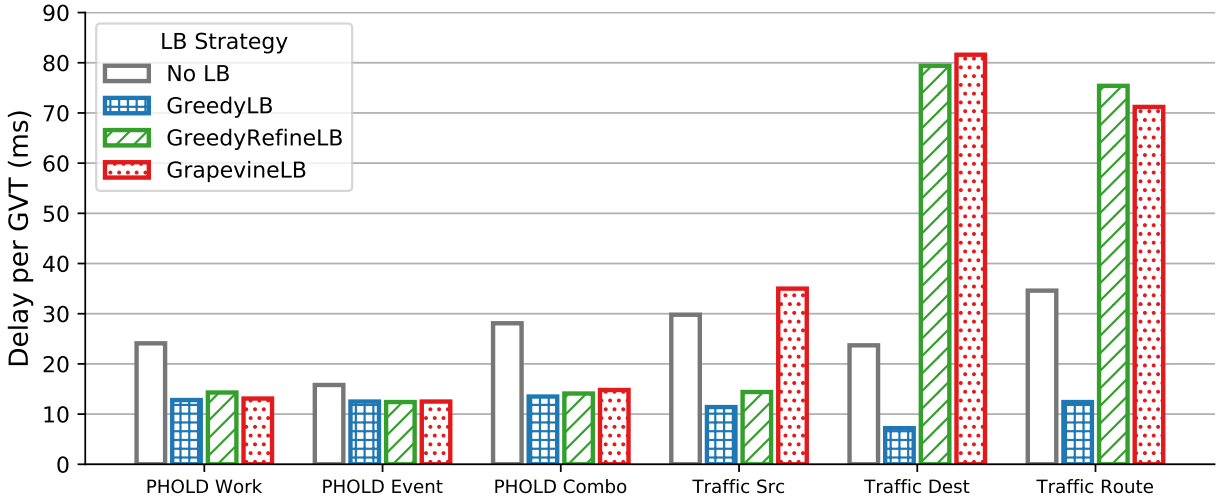

Figure 5.12: The average amount of time spent blocking per GVT computation for the leash-based GVT trigger.

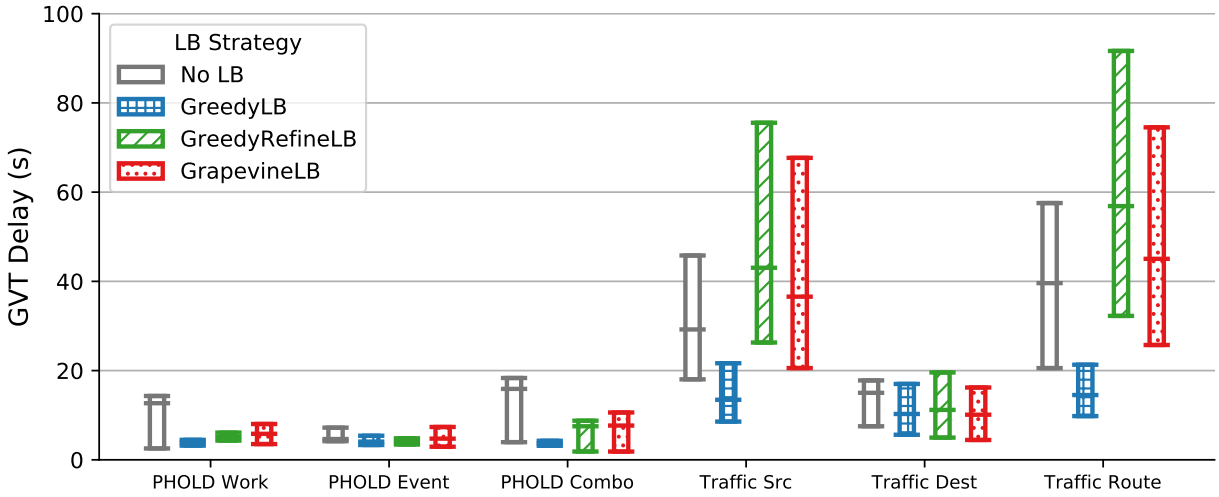

Figure 5.13: Min, max, and mean time across all processors spent blocking on the GVT computation for the count-based GVT trigger.

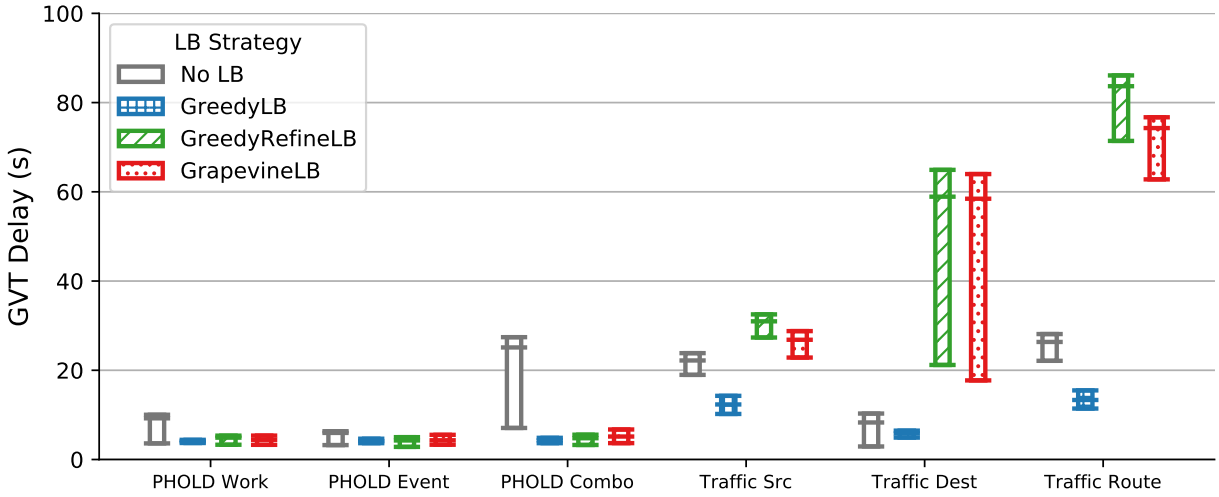

Figure 5.14: Min, max, and mean time across all processors spent blocking on the GVT computation for the leash-based GVT trigger.

much more benefit from load balancing both by lowering the max idle time, and achieving a much tighter range of times. This is largely due to the fact that one of the goals of the count-based trigger is already to maintain balance within a given GVT interval. With each process executing the same exact number of events per interval, the amount of forward work for each process will be very similar for models where the work per event is roughly equivalent. Here, the main source of variation between processes will be due to the amount of rollback and cancellation work required. The improvements to synchronization cost with load balancing under the count-based metric largely come from event efficiency improvements resulting in less rollback work. The difference for the leash-based trigger is that the amount of forward execution done is entirely dependent on the distribution of events across processes. If a given process does not have many events within a given window of virtual time it will reach the next GVT synchronization point much sooner. Load balancing helps to better distribute the amount of events per GVT interval across processes, which is why we see a more dramatic improvement to the synchronization costs with load balancing under the leash-based trigger.

5.3.4 Overhead

So far we have shown how load balancing affects overall simulation performance. Now we will explore more in depth the differences between the different load balancing strategies. As shown early on in this section, which strategy is used can have significant impact on performance, and not all strategies results in an increased event rate depending on the model being run.

When comparing different strategies, the first we will look at is how long each strategy takes to execute. Tables 5.3 and 5.4 show the total time taken by each strategy strategy on each model configuration for the count trigger and leash trigger respectively. This time includes time to gather data, execute the algorithm, and perform migrations. For the most part, decision time for each strategy is not affected by which model configuration is being run or by the GVT trigger. The exception being that GreedyRefineLB takes significantly longer for PHOLD Event with the count metric. This discrepancy will be explained when looking at migration counts. For the most part, we see that GreedyLB takes $2 - 4\times$ longer than GreedyRefineLB, and both centralized strategies take an order of magnitude longer than GrapevineLB. GreedyRefineLB runs essentially the same algorithm as GreedyLB but attempts to minimize the number of migrations, which leads to less time taken by the strategy. GrapevineLB does not require data to be gathered to a centralized place, nor does it require a single processor to look at data for every object and processor in the simulation. This results in a much quicker load balancing decision, and one that does not rely as heavily

	PHOLD			Traffic		
LB Strategy	Work	Event	Combo	Src	Dest	Route
GreedyLB	0.78s	0.80s	0.79s	0.55s	0.62s	0.55s
GreedyRefineLB	0.27s	0.86s	0.28s	0.22s	0.15s	0.23s
GrapevineLB	0.02s	0.04s	0.03s	0.03s	0.02s	0.02s

Table 5.3: Time taken by each LB strategy (with the count-based GVT trigger).

	PHOLD			Traffic		
LB Strategy	Work	Event	Combo	Src	Dest	Route
GreedyLB	0.77s	0.83s	0.77s	0.42s	0.67s	0.42s
GreedyRefineLB	0.28s	0.32s	0.28s	0.14s	0.16s	0.16s
GrapevineLB	0.02s	0.06s	0.02s	0.03s	0.03s	0.04s

Table 5.4: Time taken by each LB strategy (with the leash-based GVT trigger).

on problem size. In these experiments, load balancing was run only one time. For PHOLD this was due to the imbalance being fairly static. For Traffic, the imbalance shifts over time, however running GreedyLB multiple times caused significant overhead due to its longer execution time. Lower overhead strategies allow for more frequent load balancing on models which may require it.

Tables 5.5 and 5.6 show how many objects are migrated for each strategy. The total number of objects for each model are shown in the last row. Here we see that as expected, GreedyLB moves the vast majority of objects when it re-balances. This is due to the fact that the algorithm essentially starts from scratch where it assumes each processor is empty, and reassigns each object to processors based on weight. GreedyRefineLB attempts to accomplish a similar goal, but with much fewer migrations as we see in the table. GrapevineLB also incurs a fairly low number of migrations. In Grapevine, only overloaded processors attempt to move objects, and they only move objects until they are no longer overloaded. For the most part, number of migrations is similar for each trigger, however PHOLD Event highlights the interaction between CPU load and the triggers. As we showed earlier, for the count trigger in PHOLD Event CPU load does not capture almost any imbalance. In this case, it resulted in GreedyRefineLB behaving almost identically to GreedyLB, which caused a significantly higher number of migrations and execution time. For GrapevineLB, where the determining factor for deciding which processors attempt to move objects is max/average load ratio, no

	PHOLD			Traffic		
LB Strategy	Work	Event	Combo	Src	Dest	Route
GreedyLB	130,990	131,009	131,007	65,511	65,492	65,499
GreedyRefineLB	9,705	131,001	8,971	7,164	6,225	7,316
GrapevineLB	10,064	0	9,896	1,858	3,755	194
Total LPs	131,072	131,072	131,072	65,536	65,536	65,536

Table 5.5: Number of migrations incurred by each LB strategy (with the count-based GVT trigger). The last line shows the total number of LPs in the model.

	PHOLD			Traffic		
LB Strategy	Work	Event	Combo	Src	Dest	Route
GreedyLB	131,006	131,012	131,008	65,510	65,499	65,504
GreedyRefineLB	8,581	5,644	10,710	7,388	11,448	8,757
GrapevineLB	8,188	3,051	10,369	546	4,770	9,088
Total LPs	131,072	131,072	131,072	65,536	65,536	65,536

Table 5.6: Number of migrations incurred by each LB strategy (with the leash-based GVT trigger). The last line shows the total number of LPs in the model.

migrations were performed. This reiterates that CPU load may be an ineffective metric to determine sources of imbalance in PDES simulations.

5.3.5 Summary

In this section we looked at a number of different load balancing strategies and how they affect performance of our unbalanced model configurations. For almost all model configurations, we were able to achieve speedups when using the correct strategy. The speedups came from potential increases in event efficiency, as well as decreases in idle time due to a better balance of work. However, we also saw that the GVT trigger has a significant impact on how load balancing performs due to the ways that the different triggers constrain what work is performed. The count-based trigger tends to hide imbalances caused by event distribution which can make load balancing difficult.

Another factor which impacts the effectiveness of load balancing is the metric used to attribute load to objects. In this section we used CPU load, and the results were mixed. In

some cases, CPU load was able to capture imbalance effectively, and dynamic load balancing was able to improve performance. In other cases, CPU load was not effective in determining how to re-balance load. This was especially true in the Traffic models when using strategies which relied on aggregate processor loads to steer their balancing algorithms. In the next section we will look at a variety of other metrics which may capture different aspects of load imbalance.

5.4 METRIC COMPARISON

In the previous section we explored how load balancing can affect PDES simulations by looking at a variety of different load balancing algorithms. In many cases we saw favorable results, and in many cases load balancing was able to improve event rates by increasing event efficiency and balancing load across processors. The balance in load manifested in lower idle times waiting for GVT computations using the Blocking GVT algorithm. Strategies also exhibited different overheads in terms of decision time and objects migrated. However, using CPU time as a metric to assign load to LPs did prove problematic in some cases. For models where imbalance came in the form of event distribution, CPU load sometimes had a hard time capturing any imbalance. This was especially true for the count based GVT trigger, where each GVT cycle constrained the number of events executed per processor. This resulted in very little variation in load across processors despite the fact that there was an imbalance in number of events resulting in low event efficiencies. This was even more problematic when using strategies such as GreedyRefineLB and GrapevineLB which specifically use the amount of measured imbalance to make decisions. When the measured imbalance across processes is low due to the metric used it makes these strategies ineffective.

In this section we look at a variety of other metrics which can be used to attribute load to LPs. These metrics are described in Section 5.2 and aim to solve three potential problems that arise when using CPU time as load. First, as just described, CPU load may not actually capture imbalance in the system depending on how execution is synchronized. Secondly, the speculative execution of the TimeWarp protocol means that some of the CPU time attributed to each LP comes from incorrect execution of events and the ensuing rollbacks. It is not clear if this is desirable, and other efforts have been made to weight objects only by work that is required for the final simulation result [28, 29]. Finally, in other applications, the use of CPU time as object load is based on the expectation that behavior of the recent past is indicative of future behaviors. Namely, if an LP is executing for T units of CPU time before load balancing, we expect that after load balancing it will also require T units of CPU time. However, due to how event scheduling works in PDES simulators, this is not

necessarily the case. In some sense, the work an LP will perform in a given GVT interval is somewhat dependent on which other LPs are on the same process. This is especially true when using the count-based trigger. If a given LP has very late events compared to other LPs on the processor, it may execute very few of them. If, however, the same LP is mapped to a processor where other LPs also have very late events, or fewer events in general, then that LP will execute far more of its events. This behavior can be tricky to capture, especially when the only metric for measuring load is CPU time. It is important for the metrics we use to be able to capture imbalance in a way that is also indicative of future behavior.

We also look at what aspects of performance each metric attempts to capture. This will give important insight for how to choose effective metrics based on the characteristics of the model being simulated. By capturing snapshots of each metric before and after balancing we can also see how effective load balancing is at actually redistributing load. It is conceivable that certain metrics show similar issues to CPU time in that they are too dependent on LP mapping to be effective for load balancing.

5.4.1 Performance

Figures 5.15 and 5.16 shows the speedups of each PHOLD configuration when using the best performing strategies from the previous section and each different load metric described in Section 5.2 with both triggers. Figures 5.17 and 5.18 show the same experiments for the Traffic model configurations. All speedups were taken in comparison to runs with no load balancing. We see a wide variety of performance depending on the metric used, trigger used, and model tested. In order to better understand these results we can characterize the metrics as one of three types. First, we have the metrics that deal with virtual timestamps. These include "Current VT", "Latest VT", and "Next VT". These metrics depend on particular timestamps associated with an LP in order to make decisions on where to place it. The second category of metrics are those which use event counts. These metrics include "Committed", "Past", "Executed", "Pending", and "Active" and are simply counts of different classes of events associated with an LP. The third category is the remaining metrics that don't fit either category exactly. These metrics are CPU Time and Weighted Pending Events. CPU Time does not use event counts or virtual timestamps, and weighted pending events uses both by weighting event counts by virtual timestamps.

Next, we look at how these different classes of metrics perform as whole depending on model configuration and trigger. First we notice that, for PHOLD, metrics that relate to virtual timestamps tend to have very little effect on event rate. Because we are using the Blocking algorithm and balancing at GVT barriers, the LPs are fairly constrained in the

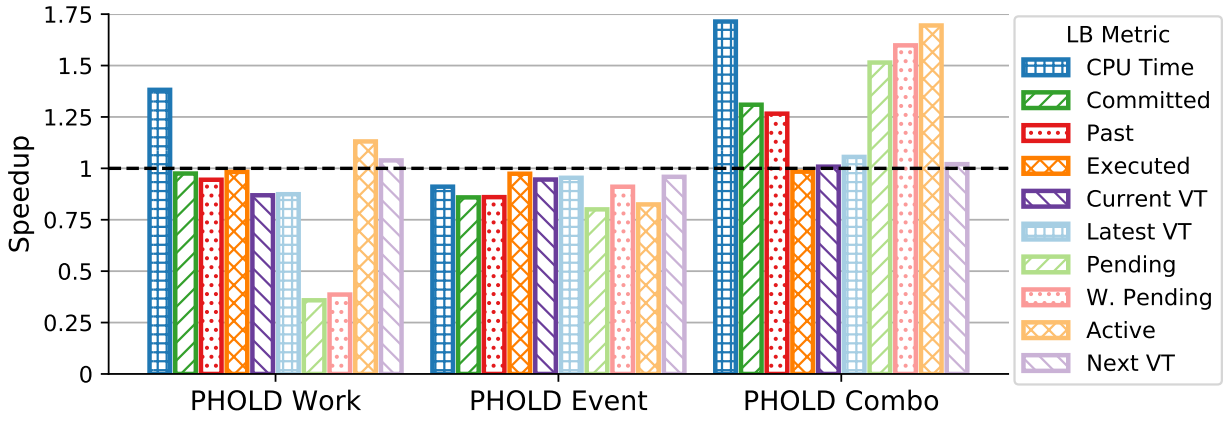

Figure 5.15: Speedup for each different load metric on each PHOLD configuration using the count-based GVT trigger.

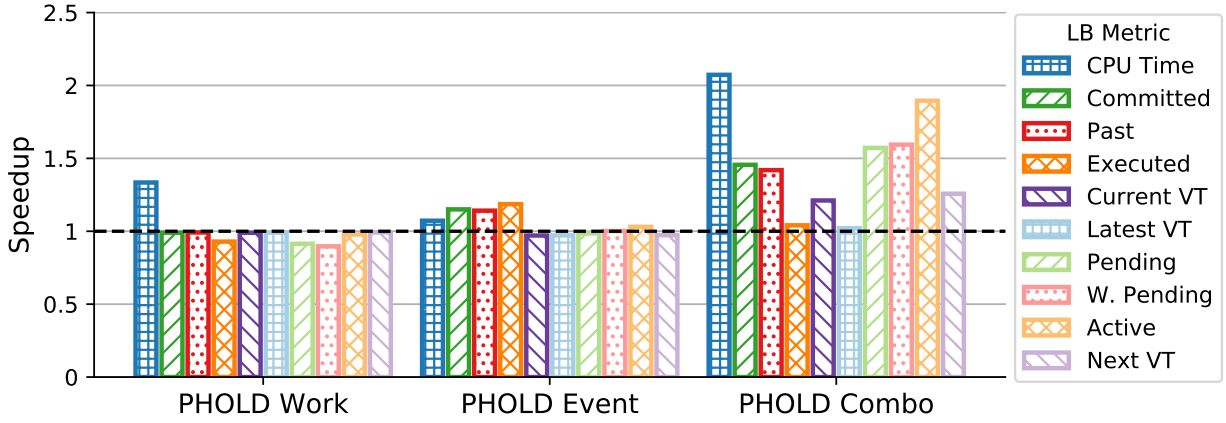

Figure 5.16: Speedup for each different load metric on each PHOLD configuration using the leash-based GVT trigger.

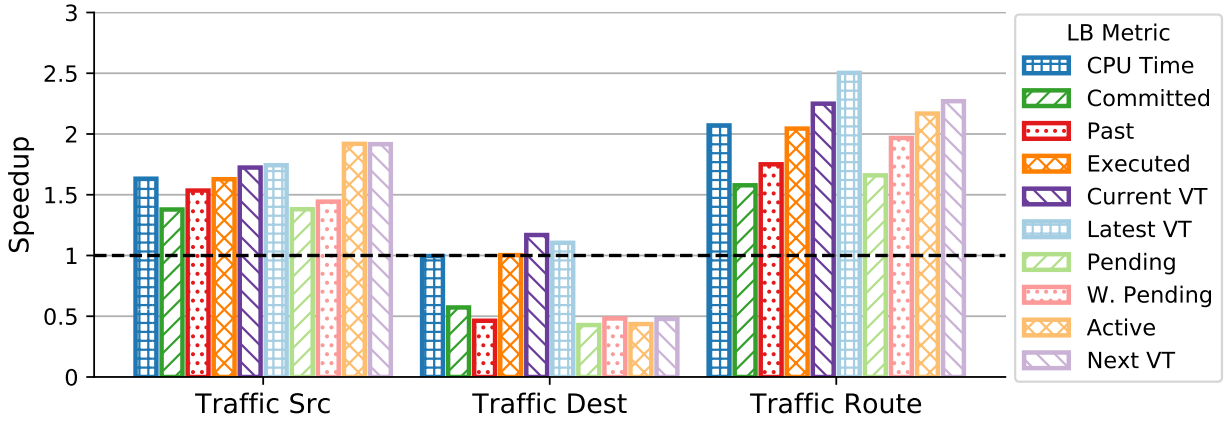

Figure 5.17: Speedup for each different load metric on each Traffic configuration using the count-based GVT trigger.

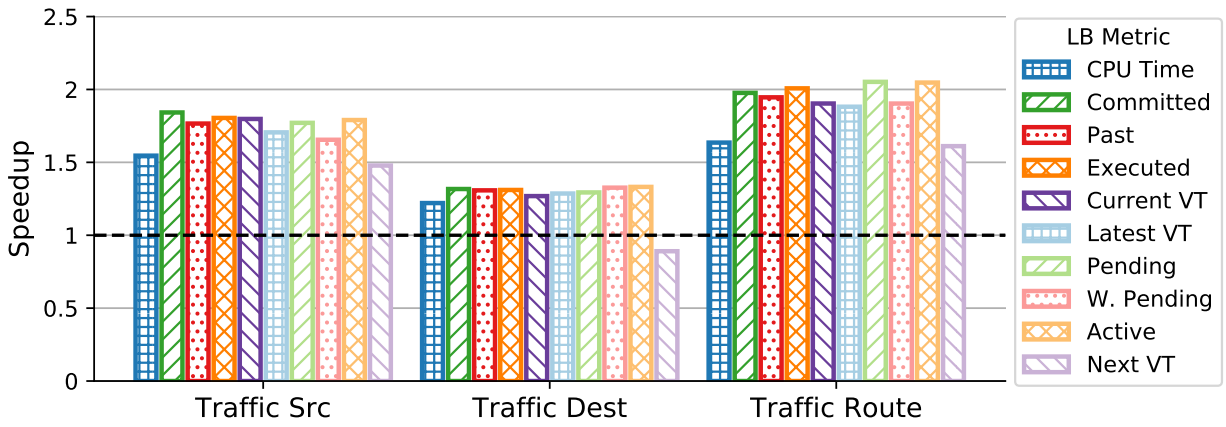

Figure 5.18: Speedup for each different load metric on each Traffic configuration using the leash-based GVT trigger.

range of timestamps they may have. This is especially true for the leash-based trigger, which literally prevents LPs from executing any events outside of the given leash. To make this clear, Figure 5.21 shows the balance of timestamps across processors for PHOLD Combo right before balancing occurs using the leash-based GVT trigger. In these cases, there is very little variation across processors which gives the load balancer almost no way to differentiate when deciding how to move load. However, we will revisit these metrics in the next section when looking at load balancing with non-blocking GVT algorithms. For Traffic, we do see more improvement when using the timestamp based metric with the count trigger. Since the count trigger does not constrain LPs within a leash and the distribution of events is more unbalanced in Traffic, using timestamps can be an effective metric for assigning load. This is largely due to the lower event efficiency of the Traffic model. Even though each processor executes similar numbers of events before load balancing, high numbers of rollbacks on certain processors cause large variations in timestamps. Figure 5.20 shows these variations for the Traffic Route configuration using the count-based GVT trigger. This variation gives the load balancing strategies the information needed when deciding how to move objects around to even out the rates of progress across processors. In particular, the latest virtual time and the next virtual time are two of the best performing metrics for both Traffic Src and Traffic Route. As seen in Figure 5.17 both yield over 2× speedup in Traffic Route.

The second class of metrics – those based on event counts – show more favorable performance in PHOLD models. With the exception of PHOLD Work, each other model configuration has some imbalance in how events are distributed across LPs. These metrics based on event counts are able to capture these imbalances more directly and more effectively than relying on CPU load. For PHOLD Event, the best performing metrics are Committed

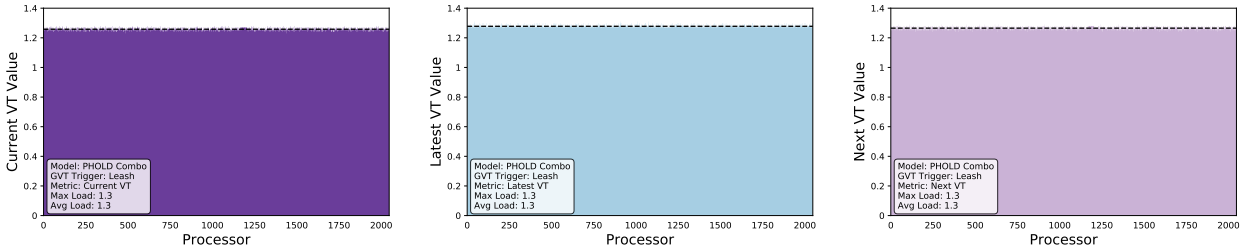

Figure 5.19: The distribution of load values across processors for each of the three timestamp based metrics for PHOLD Combo using the leash-based GVT trigger. The Blocking GVT algorithm constrains the progress of LPs resulting in a very balanced distribution of timestamps despite other potential sources of imbalance.

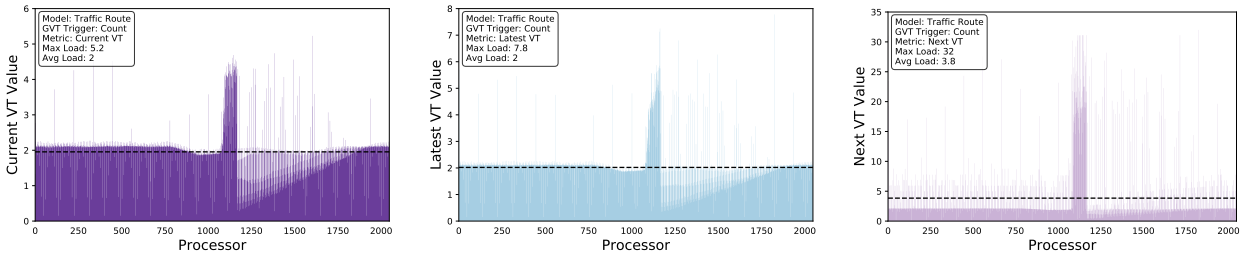

Figure 5.20: The distribution of load values across processors for each of the three timestamp based metrics for Traffic Route using the count-based GVT trigger. Due to low event efficiency, processors progress through virtual time at different rates, which these metrics capture and expose to the load balancing strategies.

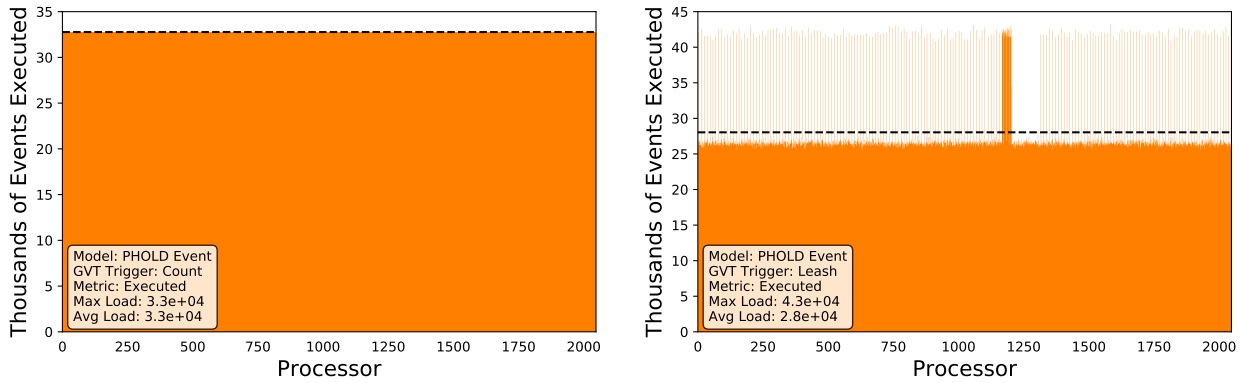

Figure 5.21: The distribution of total events executed by each processor in the 2,048 processor run for PHOLD Event with the count-based GVT trigger (left) and the leash-based GVT trigger (right). The type of GVT synchronization heavily effects how imbalance manifests.

Events, Past Events, and Executed Events, where Executed Events gets 1.2× speedup with the leash GVT trigger. Considering the leash trigger was already almost 1.5× faster than the count trigger, load balancing based on Executed Events provides the fastest configuration of PHOLD Event when using the Blocking GVT algorithm by a pretty good margin. However, again we see that the blocking nature of the GVT can have a large effect on how this imbalance manifests. Figure 5.21 shows the number of executed events for PHOLD Event right before balancing under each GVT trigger. The count-based trigger essentially hides the imbalance entirely which is why Figure 5.15 shows no speedup. The leash-based trigger however is able to highlight the fact that there is an imbalance in the amount of events executed per time interval, which leads to the 1.5× speedup achieved. For Traffic, these metrics are not quite as impressive. Especially when looking at the count triggered GVT, the metrics based on event count are consistently outperformed by virtual time based metrics. For Traffic, the rate of progress of LPs is a more important factor for balancing load, and that is captured better using virtual timestamps. When the event efficiency is so low, as it is for Traffic when using the count-based GVT trigger, simply counting events from an LPs past does accurately account for all of the work and irregularity caused by the high amount of rollbacks. Because of this, we also see that the metric based on total number of active events tends to be almost as effective as the timestamp based metrics. By looking at the number of events active, it estimates future work by considering both forward execution and the number of potential rollbacks that may occur. When event efficiency is low, both of these factors will heavily contribute to the work an LP will be doing in the near future.

5.4.2 Event Efficiency

As in the previous section, we want to see how each metric effects event efficiency. Figures 5.22 and 5.23 show the event efficiency for each PHOLD configuration and metric. Here we can clearly see that not all metrics have a positive effect on efficiency. We also see that the speedups from previous plots almost directly mirror the event efficiency plots. As with the last section, event efficiency is less effected when using the leash metric. In all cases, the highest event efficiency is actually achieved using the CPU time metric to attribute load. This makes sense considering in almost all cases, CPU time also saw the highest speedups for PHOLD.

For Traffic, shown in Figures 5.24 and 5.25 event efficiency is much worse than for PHOLD. For the count-based trigger, event efficiency shows the most improvements when we balance based off of virtual timestamp for Traffic Src and Traffic Route. Again, the speed at which LPs moves through virtual time is the key factor in these configurations. For the leash

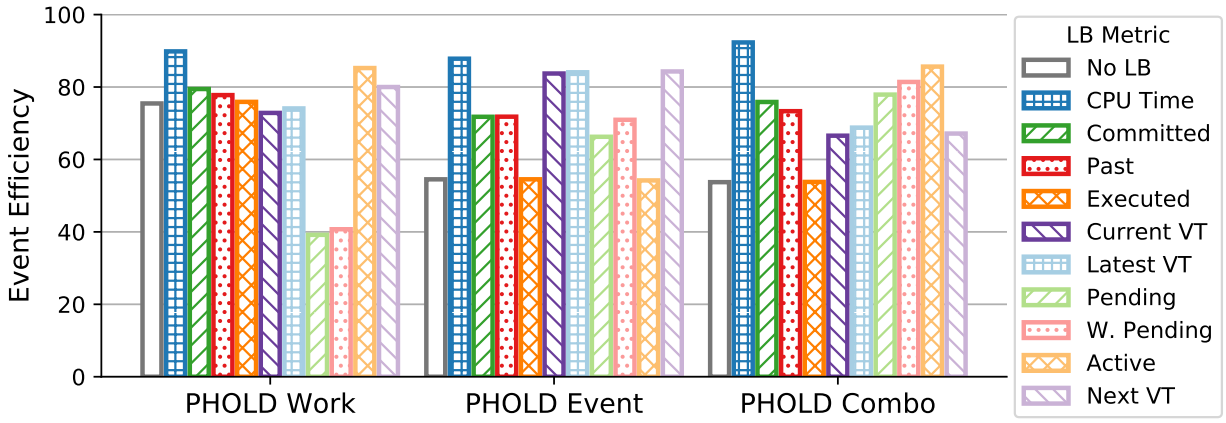

Figure 5.22: Event efficiency for each load metric on each PHOLD configuration using the count-based GVT trigger.

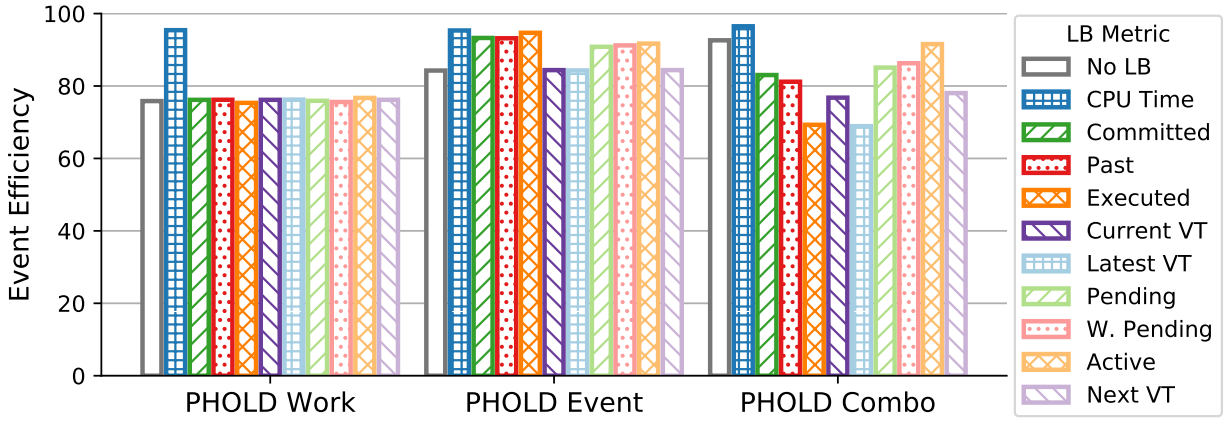

Figure 5.23: Event efficiency for each load metric on each PHOLD configuration using the leash-based GVT trigger.

metric, the event efficiency is actually lower when load balancing is used in all cases. As in the last chapter, much of the speedup comes from improved load balance manifested in more balanced GVT delays.

5.4.3 Synchronization Cost

Figures 5.26 and 5.27 show the ranges of time spent waiting on the GVT computation for the count-based and leash-based GVT triggers respectively. For all configurations other than Traffic Dest using the count-based trigger there is a significant decrease in min, max, and average time each processor spends blocking on the GVT. In most cases the range of times spent blocking is also significantly smaller. This improvement is what accounts for the

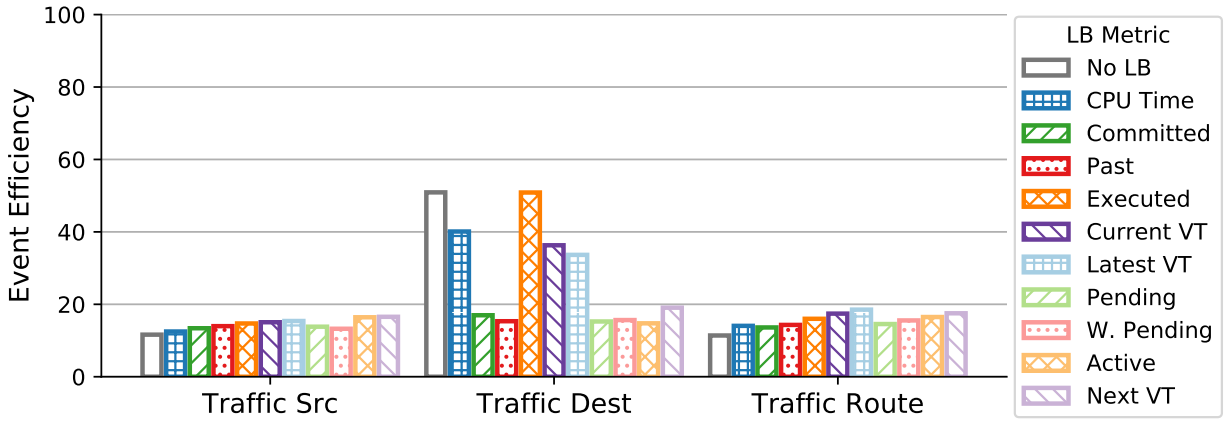

Figure 5.24: Event efficiency for each load metric on each Traffic configuration using the count-based GVT trigger.

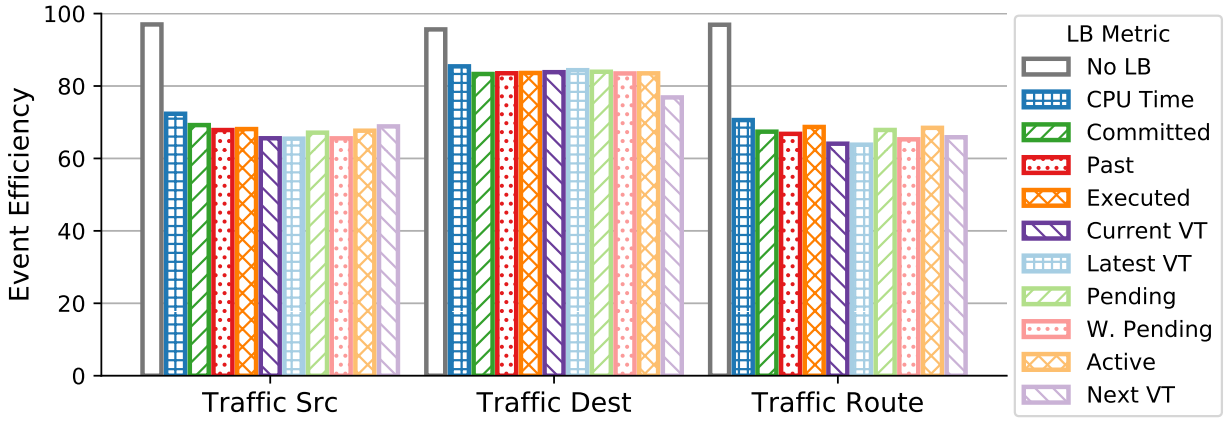

Figure 5.25: Event efficiency for each load metric on each Traffic configuration using the leash-based GVT trigger.

significant speedups in the Traffic model, despite the fact that event efficiency does not see significant improvement. Specifically looking at the count-based metric we see similar trends as before where metrics which rely on event counts of just past or just future events tend to do poorly because of the low event efficiency. The metrics which are based on timestamps or take into account both past and future event counts do a better job at reducing the amount of idle time from the GVT.

5.4.4 Summary

In this section we explored a number of different metrics for attributing load to LPs. CPU time is a commonly used metric in a wide variety of other HPC applications. Using the

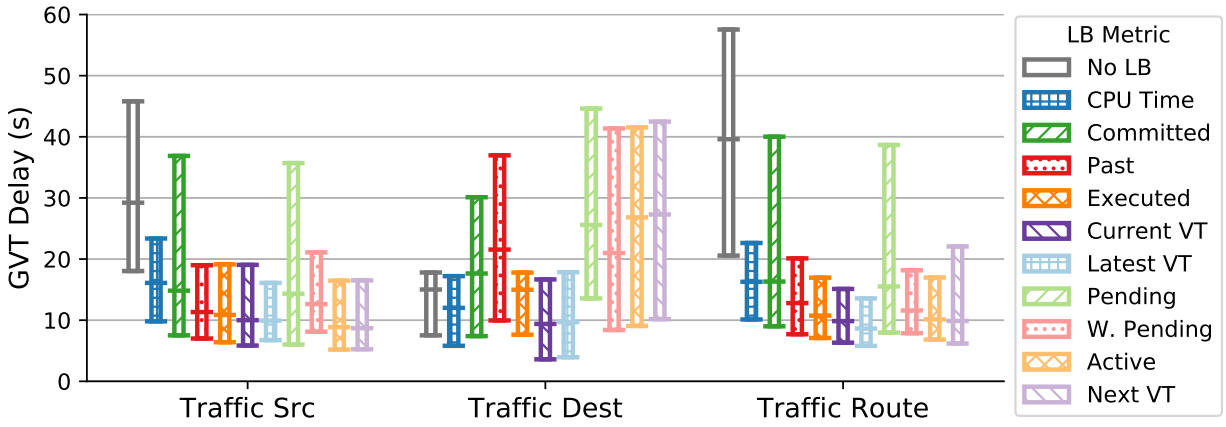

Figure 5.26: The range of min, max, and average GVT delay across all processors for each load metric and each Traffic configuration when using the count-based GVT trigger.

number of committed events to attribute load to LPs based only on work that is part of the final simulation result is a common practice in other PDES simulators [28, 29]. However in this work we showed that in certain cases, these metrics are not necessarily the best choices when attempting to balance speculatively executing simulations.

In the PHOLD model configurations, as well as Traffic configurations which exhibit a high event efficiency, using CPU time does often provide an accurate estimate of an LPs work. This is due to the fact that most of the work that occurred before load balancing was forward execution work. Therefore, CPU Time does provide an accurate measure of meaningful work performed, and imbalances in that work can be effectively used to re-balance. This also holds true for metrics which are based on event counts for events which have already been executed. All of these metrics take some estimate of work that was previously done, whether it be measured in CPU time or numbers of events, and use that as an estimate of an LPs future work. If event efficiency is high, this works well as the simulation works similarly to more traditional applications which do not rely on speculative execution.

However, most of the Traffic model configurations suffer from very low event efficiency. Here, relying on work done in the past may not accurately indicate imbalances in the model, as a large portion of that work is incorrect execution or rollbacks. This leaves us with two new options: looking at virtual timestamps, or future events. Using various metrics based on virtual timestamps performed best for these Traffic configurations because they more accurately capture the imbalances in rate of progress of LPs. They attempt to balance rate of progress through virtual time across processes, which can lead to modest improvements in event efficiency and large decreases in idle time due to GVT imbalances.

Looking at future events does not lead to improvements for the same reason that past

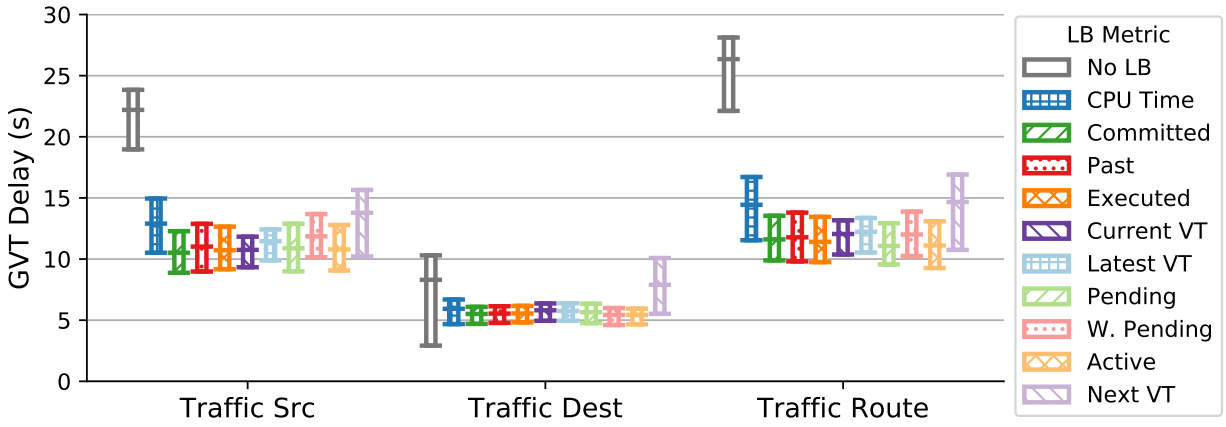

Figure 5.27: The range of min, max, and average GVT delay across all processors for each load metric and each Traffic configuration when using the leash-based GVT trigger.

looking metrics were not effective. With such low event efficiency, there is little likelihood that many of these future events will even be executed. However, hybrid metrics which look at all active events (both pending and processed) do a good job of estimating all work an LP will doing in the future. This is due to the fact that any work an LP will do is either going to be executing a pending event, or rolling back a processed event.

Finally, the original goal of this section was to use load balancing to increase event efficiency. However, in many experiments we saw that the best performance came from configurations with lower or equivalent event efficiencies. In these cases, performance increases came from the fact that the metrics were able to capture some notion of event inefficiency and use that to balance workloads despite the poor event efficiency.

5.5 COMBINING WITH NON-BLOCKING GVT

So far, when looking at the effects of load balancing we have focused on its use in conjunction with the Blocking GVT algorithm. This allowed us to get some insight into how balancing affected various parts of the simulation, as well as how simulation configuration had an impact on load balancing. In a number of the configurations studied, we saw that load balancing was able to increase event efficiency by creating a more favorable distribution of LPs to processors. In other cases, performance improvements came from a better balance of work per GVT interval across processes. We were even able to balance loads which were highly speculative due to very low event efficiencies by choosing the correct load metrics. This becomes particularly important when considering the fact that in the later sections of Chapter 4 we saw that non-blocking GVT computations tended to suffer from

low event efficiency. They still managed to improve overall performance by drastically reducing synchronization costs, but the low event efficiency gave a clear avenue for potential improvement. In this section we show that using dynamic load balancing we can further improve performance of non-blocking GVT algorithms.

In this section, we will be combining dynamic load balancing with the non-blocking GVT algorithms studied in Chapter 4. We first begin by looking at the Phase-Based GVT algorithm with blocking load balancing. Because the Phase-Based algorithm computes exactly one GVT per complete run of the algorithm it makes it simple to add blocking load balancing at the completion of the GVT computation. When the Phase-Based algorithm is run during the GVT interval on which we want to perform load balancing, event execution does not resume as it normally does after Phase-Based GVT computations. Once the GVT is computed, load balancing occurs, and then after migrations have completed the simulation is resumed. We then look at fully non-blocking load balancing using the Adaptive Bucketed GVT algorithm. For these experiments, load balancing is still triggered after a GVT update, but the simulation is able to run continuously during the GVT computation and load balancing. These experiments also utilize the adaptive throttling from Section 4.4.3. The Adaptive Bucketed algorithm with adaptive throttling achieved the best performance out of all non-load balancing configurations in this thesis. The potential of further improving this method by adding load balancing is an attractive option.

5.5.1 Phase-Based GVT Load Balancing

Figure 5.28 shows speedups for PHOLD configurations when using the Phase-Based GVT algorithm with load balancing, compared to runs using the Phase-Based algorithm but no load balancing. As before, speedups are computed using the entire runtime of the simulation. In these experiments, GrapevineLB was used instead of GreedyRefineLB. As we saw in Section 5.3, GreedyRefineLB performed best when the Blocking GVT algorithm was used, but in this case the extremely low overhead of GrapevineLB had slightly better performance. Figure 5.29 shows the associated event efficiencies with these experiments. First, with PHOLD Work, we see as before that CPU time is the only effective load metric. Again this is due to the fact that the imbalance in PHOLD Work is entirely based on CPU time required to execute events. The CPU load metric achieves 1.35× speedup over the run without load balancing, and also increases event efficiency from 50% to 80%. In this case, we are able to reap the benefits of the low synchronization cost of the non-blocking GVT, while still also achieving a relatively high event efficiency. This results in the best performing PHOLD Work configuration out of all experiments in this thesis. However, since

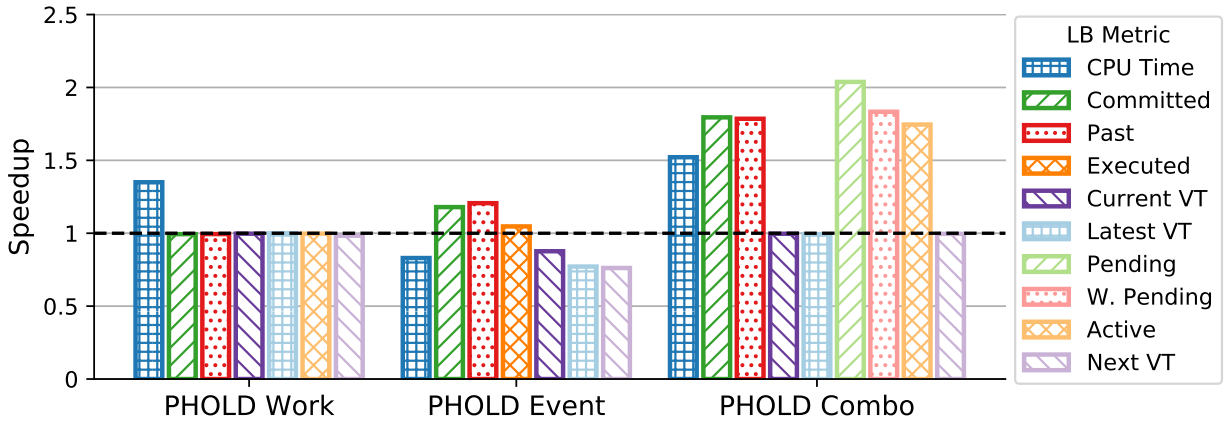

Figure 5.28: Speedup of each PHOLD model configuration and metric when doing load balancing in conjunction with the Phase-Based GVT algorithm. Speedups are taken over a full run of the best Phase-Based configuration with no load balancing from Chapter 4.

PHOLD Work is still entirely balanced in terms of distribution of events across LPs, the metrics which utilize event count to detect imbalance show no effect. As we will see with each PHOLD configuration, there are some bars not shown (Executed for PHOLD Work and Combo, Pending and W. Pending for PHOLD Work and PHOLD Event, and Active for PHOLD Event). These metrics, when combined with load balancing and the continuous GVT perturbed the simulation in such a way that it would hang or crash due to cascading event cancellations and rollbacks. When the simulation runs unimpeded by blocking from the GVT computation, some metrics can become so unbalanced that re-balancing based on these metrics results in poor performance.

PHOLD Event shows a very different story from PHOLD Work. CPU time results in worse performance than no load balancing, as do metrics based on virtual time progress. The distribution of these metrics does not accurately capture imbalances in the model, and load balancing does not have much impact other than adding overhead to block and run the strategy. However, both Committed Event count and Past Event count show around 1.2× speedup. For this case, balancing based on recently executed events proves to be an effective metric, as the imbalances in PHOLD Event primarily stem from an uneven distribution in events to LPs. This is directly captured by both of these metrics, and load balancing is effectively redistributing the number of events executed on each processor. This results in an improved event efficiency by about 10%. Again, the higher event efficiency coupled with low synchronization cost is able to get the best performance out of all studied configurations for PHOLD Event.

Finally, PHOLD Combo contains both forms of imbalance, and stands to gain the most

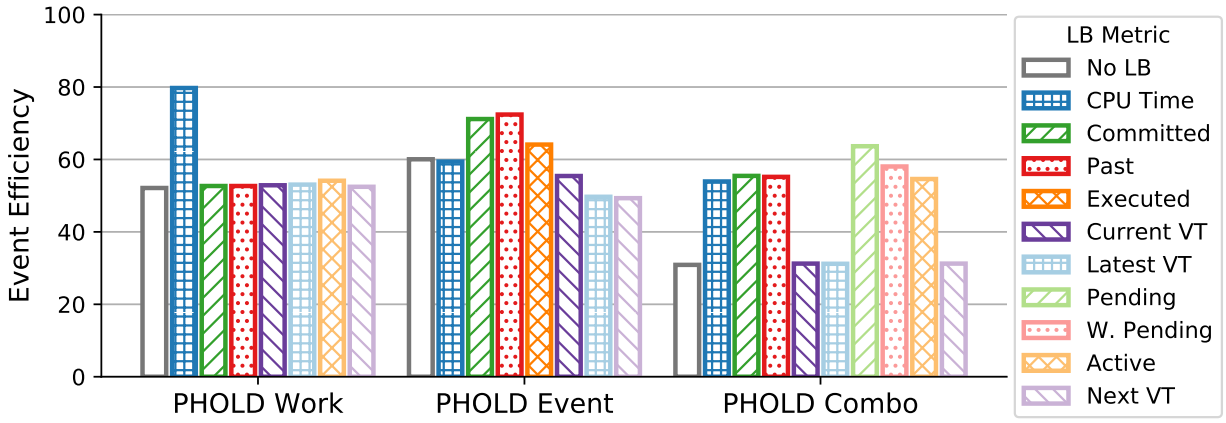

Figure 5.29: Event efficiency of each PHOLD model configuration and metric when doing load balancing in conjunction with the Phase-Based GVT algorithm.

benefit from dynamic load balancing. The speedups shown in Figure 5.28 back this up by showing speedups for a large number of metrics. Using the number of Pending Events as a metric results in slightly over 2× speedup, again providing the best performance out of all PHOLD Combo experiments. Similarly, Figure 5.29 shows event efficiency is increased for most metrics by up to 20%.

These results show clear evidence that load balancing can be used to increase event efficiency independently from the GVT computation. This increases the potential of non-blocking GVT algorithms, which are often limited by a low event efficiency due to unbounded optimism. However, in certain cases the imbalance or event efficiency can become too extreme for load balancing to deal with effectively. There were certain metrics which failed to run successfully for PHOLD, and for Traffic we were unable to run load balancing in conjunction with the Phase-Based algorithm at all. The extremely low event efficiency of Traffic in this setup, along with its strong communication locality caused the simulations event cancellations and rollbacks to cascade out of control when load balancing was introduced. However, as shown in Section 4.4.3, adaptive event throttling in the Adaptive Bucketed GVT algorithm was able to increase event efficiency for Traffic models while still allowing for continuous execution of events. This provides the motivation for our next set of experiments.

5.5.2 Adaptive Bucketed GVT Load Balancing

Figure 5.30 shows the speedups achieved when adding load balancing to the Traffic model using the Adaptive Bucketed GVT algorithm. As in the previous experiments, speedup is

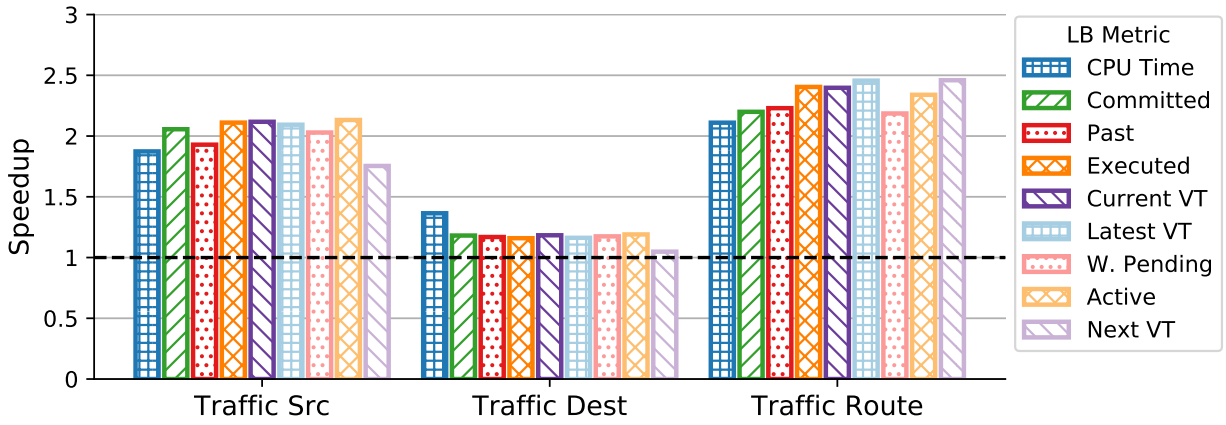

Figure 5.30: Speedup of each Traffic model configuration and metric when doing load balancing in conjunction with the Adaptive Bucketed GVT algorithm with event throttling. Speedups are taken over a full run of the best Adaptive Bucketed configuration with no load balancing from Chapter 4.

taken over the entire runtime of each simulation. In this case we see speedups in every single configuration, with 2.5× speedup achieved for PHOLD Route with both the Latest VT and Next VT metrics. Furthermore, these experiments were significantly more resilient to failures due to the adaptive throttling, as only the Pending Events metric failed to run successfully. Like in the previous section, Traffic Src and Traffic Route both get the most benefit from timestamp based metrics, and the active events metric. However many of the other metrics are very close in performance as well. As each LP can run unimpeded, there is more room for imbalances in both event counts and virtual time progress to manifest. Furthermore the adaptive throttling increases the baseline event efficiency a bit so that we avoid some of the issues in balancing low event efficiency models that showed up in the previous section. Figure 5.31 shows the event efficiencies for the various configurations. For Traffic Src and Traffic Route, event efficiency nearly doubles for most metrics. Event efficiency for Traffic Dest is actually lower, however we still see event rate improvements coming from a more balanced workload across processes.

These speedups are especially impressive in context of the rest of the results from this thesis. The Adaptive Bucketed algorithm configurations from Chapter 4 were already the best performing configurations for each Traffic model configuration when no load balancing was used. However, even in terms of configurations for Traffic where load balancing was used, these results achieve a higher event rate by a factor of 1.2× for Traffic Src, 1.1× for Traffic Dest, and 1.3× for Traffic Route. The combination of the Adaptive Bucketed algorithm, adaptive event throttling, and dynamic load balancing are able to reduce synchronization

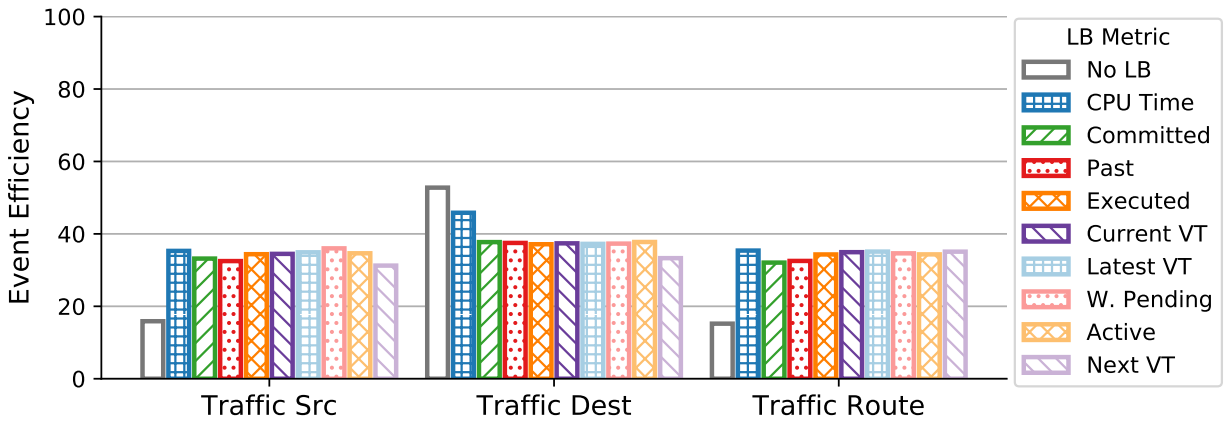

Figure 5.31: Event efficiency of each Traffic model configuration and metric when doing load balancing in conjunction with the Adaptive Bucketed GVT algorithm.

costs, increase event efficiency, and balance work across processes respectively. With the original experiments from this thesis, event efficiency, synchronization cost, and balance of work were all tightly coupled by the configuration of the Blocking GVT algorithm. Adjusting the GVT interval lowered both the synchronization cost and event efficiency in tandem. Adjusting the type of GVT trigger also affected this performance characteristics, while also affecting the potential balance of work across processors per GVT interval. This made tuning model performance difficult. Here, by decoupling these concerns, each performance characteristic is able to be addressed by an independent piece of the simulation.

5.5.3 Summary

In this section, we combined techniques from throughout this entire thesis by using dynamic load balancing together with non-blocking GVT algorithms. Previously Chapter 4 showed that non-blocking GVT algorithms demonstrated superior performance to blocking algorithms, but with the drawback of low event efficiency. Load balancing demonstrated the ability to improve simulation performance by balancing load as well as increasing event efficiency. By combining these two approaches, we were able to gain the benefits of low synchronization costs for the GVT algorithm, while still achieving relatively high event efficiencies. This resulted in the highest event rates for each PHOLD model configuration in this thesis.

However, for particularly low event efficiency models such as Traffic, balancing in the context of non-blocking execution was too brittle and led to cascading event cancellations and rollbacks. By adding in the adaptive event throttling capabilities of the Adaptive Bucketed

GVT algorithm, we increased the event efficiency enough to make load balancing more robust and effective. The decoupled components of this process allowed the GVT algorithm to reduce synchronization cost, the throttling to manage event efficiency, and the load balancing to balance load and ensure all processors still had enough work to execute continuously. This resulted in the highest event rates for every Traffic model configuration across the entire thesis.

5.6 CONCLUSION

This chapter focused on the use of dynamic load balancing to improve performance of optimistic PDES simulations. In particular, it allows the simulator to better adapt to model conditions during execution by remapping LPs. We showed that this remapping can affect model performance by improving the balancing of work across processors, as well as increasing event efficiency. The latter is a benefit of load balancing which is somewhat unique to PDES due to the speculative execution present in optimistic simulations.

We also analyzed a large number of metrics for attributing load to LPs. CPU time and committed event count have both been used by other works in HPC and PDES to great effect. However, our results show that the metric which best captures imbalance can vary widely based on simulator configuration and which model is being ran. In some cases the best performing metrics did not rely on event counts, but rather on virtual timestamps associated with each LP. This was especially true for models with very low event efficiencies, where simply taking into account past work was a poor indicator of work an LP would be doing in the near future.

Finally, based on observations from both the load balancing and GVT experiments, we combined dynamic load balancing with non-blocking GVT computations. In our earlier experiments, non-blocking GVT computations performed well despite suffering from a low event efficiency. Dynamic load balancing was shown to be able to improve event efficiency through a more effective mapping of LPs to processors. By combining the two we were able to demonstrate clear evidence that load balancing could be used to further unlock the potential of non-blocking GVT algorithms. This resulted in the best performance for each model configuration out of all experiments run in this thesis.

CHAPTER 6: CONCLUSION

There are two schools of thought in Parallel Discrete Event Simulation when approaching how to synchronize event execution across distributed processes. Conservative synchronization protocols maintain causality constraints by requiring that each event can be guaranteed safe to execute before executing it. Optimistic synchronization protocols allow events to be executed freely as long as the simulator can recover from causality violations. Traditionally, optimistic protocols are an attractive option because they give simulators a high degree of freedom to exploit parallelism. However, they may also add a significant amount of complexity to the simulator and model execution. Dealing with this extra complexity can add significant overheads that manifest differently for different models and configurations. These overheads can drastically reduce the benefits gained from a high degree of parallelism and can make models difficult to configure and run effectively.

In this thesis, we demonstrated a number of ways to alleviate some of the complexity of optimistic PDES in order to more effectively reap the the benefits of its highly parallel execution. Each technique we looked at focused on allowing the simulator to dynamically adapt to simulation conditions in order to improve performance. Being able to adapt dynamically is particularly important due to the fact that a simulator may need to run a wide variety of models, each with varying characteristics that can impact performance. Specifically, we looked at message-driven execution to adaptively schedule event communication and computation alongside other simulator work. We developed a new GVT algorithm which utilizes timestamp information to adapt to simulation conditions, allowing it to update the GVT without blocking event execution or requiring large amounts of collective communication. Finally, we analyzed effective load balancing techniques to migrate LPs on the fly to achieve a better balance of load and higher event efficiency.

6.1 MESSAGE-DRIVEN EXECUTION

First, in Chapter 3 we demonstrated how message-driven execution is a highly effective execution model for PDES simulations. Charm++ was used as our particular implementation of a message-driven execution environment and runtime system. PDES simulations are often characterized by a large amount of very fine-grained communications. This communication is primarily due to sending events between LPs located on different processors. Simulations may require millions of events or more to be sent, and each event often has a very small amount of data to carry with it. These sends are spread over the entire simulation

and are inherently asynchronous and one-sided; once an LP creates and sends and event it requires no response and can immediately continue execution. Furthermore, all of the work an LP will perform for a given simulation is entirely dictated by the events scheduled for it. Because of this, using message-driven execution within a PDES simulator yields a number of important benefits. It provides a very simple and natural implementation, where LPs are parallel objects, and events are the asynchronous messages sent between them. Our simulator had a much smaller and more modular code base than an equivalent implementation using a more process oriented execution model.

More importantly, the message-driven model also resulted in significant performance improvements to the models we experimented with. The primary reason for this was that the execution model allowed the simulator to better adapt to the communication and computation patterns of models it ran. The runtime system scheduler adaptively scheduled the communication and computation required by the simulator, which significantly cut down on communication overheads. The latency of fine-grained message sends was hidden by overlapping their communication with execution of other available events. Another important aspect of the adaptive scheduling is that it also allows other simulator work and communication to be seamlessly and automatically overlapped with event execution. This allowed a high degree of flexibility in implementing and experimenting with various GVT algorithms. This leads directly into the next contribution of this thesis: improvements to the GVT computation.

6.2 GVT COMPUTATION

Chapter 4 explores a number of different GVT algorithms and gives an in-depth analysis of their effects on key simulation characteristics. We first look at the very basic Blocking GVT computation, and show the concrete ways in which different configurations affect synchronization costs, event efficiency, and overall simulation performance. This analysis is used throughout the rest of the work as a basis from which to evaluate other techniques. It also reveals a strong coupling between event efficiency and synchronization costs, and shows the importance in achieving a good tradeoff between these two quantities. As the thesis progresses we aim to decouple these two quantities from each other so that each can be treated separately.

We then show an effective and completely non-blocking implementation of Mattern's Phase-Based GVT algorithm [19] which again relies on message-driven execution to work effectively. Our implementation allows the `GVTManager` and `Scheduler` to each perform their own communication and computation, while allowing the runtime system to dynami-

cally schedule each piece. No special communication routines needed to be written, and no changes to the `Scheduler` are required to run alongside the `GVTManager`. By not requiring event execution to block during the GVT computation, synchronization costs are effectively reduced to zero and most models and configurations saw significant speedups. However, two problems still existed. First, event efficiency was low due to the fact that there was no longer a bound on the simulators optimism. Second, because the algorithm was not aware of any causality information, we often saw an increase in the number of GVT computations required to complete a simulation. This also sometimes led to difficulty in configuring models with high memory consumption.

Here we also propose a new GVT algorithm, the Adaptive Bucketed GVT algorithm. The algorithm is able to run almost completely independently from the simulation `Scheduler`, and never blocks event execution. Furthermore, it incorporates timestamp information from sent and received events to adapt to the flow of the simulation. This provides two important benefits. First, once the algorithm begins, newly sent events can still be incorporated into the computation. With the Phase-Based GVT algorithm, once it began computation all newly sent events would not be incorporated until the next GVT computation. Secondly, it allows the algorithm to adaptively expand or contract the virtual time window it encompasses based on the progress of each processor. For models tested, this results in far less collective communication required by the GVT algorithm. The algorithm still resulted in a low event efficiency similar to the Phase-Based algorithm. However, it also presented an avenue for dealing with this problem by using information already collected in the process of the computation. By allowing the `GVTManager` to selectively throttle certain events, we were able to significantly improve event efficiency while also lowering the total amount of events and anti-events sent between remote processors. In certain model configurations, this resulted in higher event rates. It also became an important part of the work done in Chapter 5.

6.3 DYNAMIC LOAD BALANCING

In Chapter 5 we looked at dynamic load balancing as another way in which our simulator could adapt to complex model workloads to improve performance. We started by evaluating a number of different load balancing algorithms on each model. In starting with the Blocking GVT algorithm, we showed that load balancing effectiveness is extremely dependent on the trigger for determining when to compute the GVT. Whether the trigger was a specific event count or a virtual time window had significant impact on what work had been completed when load balancing began. As a result the trigger also had significant impact on the loads

which were presented to the load balancer, and affected how the load balancer chose which objects to migrate. This also had an impact on which characteristics were most affected by load balancing. When balancing using event count, load balancing had less effect on the balance of work to processors, but a more significant effect on event efficiency. When using a virtual time window, the opposite held true.

We next explored a variety of load metrics which could be used to attribute load to LPs. In traditional HPC applications, CPU time is used to assign load to objects. This is also what we used in our initial experiments. However, speculative execution means that CPU time also includes erroneous event executions and rollback work as part of an LPs load, which intuitively seems problematic. A common solution to this in other PDES load balancing work has been to use the number of committed events to represent the load of each LP. We evaluated this along with a large number of other metrics and analyzed the results. We found that, as expected, CPU time often performed poorly compared to PDES specific metrics. However, we also found that although committed events did sometimes prove to be more effective than CPU load, there were a number of other metrics which performed even better. The problem with committed events is that it ignores incorrect work, and may also not be indicative of future workloads when LP mapping changes. Even though incorrect work and subsequent rollbacks are not part of the final result, they still affect how long each processor takes to reach the next GVT computation. With blocking GVTs, an imbalance in that time can result in more overhead and more time spent idle. Here we showed that metrics which more directly capture total work, future work, or rate of progress can sometimes provide an even better metric to capture imbalance.

Finally, we showed that combining dynamic load balancing with the non-blocking GVT algorithms from Chapter 4 provides the best performance out of all experiments performed in this thesis. First, we showed that load balancing was able to increase event rate by increasing event efficiency when running the PHOLD model using the Phase-Based GVT algorithm. The Traffic model, however, had such a low event efficiency that load balancing had difficulty having a positive impact on performance. Based on these observations, we combined the Adaptive Bucketed GVT algorithm, adaptive event throttling, and dynamic load balancing to run the Traffic model. By completely decoupling concerns, we allow the event throttling to handle event efficiency, load balancing to handle mapping of work load, and the GVT algorithm to keep synchronization costs low. This yielded significant performance improvements for Traffic, and outperformed every other scenario from this thesis.

6.4 FUTURE WORK

There are a few open questions raised by the contributions in this thesis:

First, in the Adaptive Bucketed GVT algorithm, we used offline analysis of event statistics to tune how the algorithm selects events to hold back. It showed to be effective at decreasing communication and increasing event efficiency. However, it had to be tuned to each model despite the fact that data collected during execution could be used to make the decisions online. By using something like machine learning to allow the `GVTManager` to identify patterns in this data, the algorithm could be even more effective in improving performance in a model independent fashion. It would also be able to change these decisions over the course of a simulation, and make different decisions per process, instead of the current static decision used by every single process in the simulation.

Second, we presented a large number of metrics which captured load imbalance in different ways. The effectiveness of each metric varied based on model and simulator configurations. A load balancing framework which could more intelligently determine which metric to use, or one which could use a vector of multiple metrics could prove useful. This would allow the simulator to more effectively adapt to a variety of different models without as much manual tuning.

Finally, our original hypothesis was that to get effective load balancing we could focus solely choosing good load metrics. However, the experiments in this thesis provided evidence that PDES specific load balancing strategies may also be needed. In many cases, the strategies which relied on max to average load ratio to drive decision making did not detect meaningful imbalance. However, in doing a full re-balance, GreedyLB was still able to improve performance. It is possible that for PDES, strategies may need to employ different algorithms to determine which objects to move. Furthermore, models like Traffic have a high degree of communication locality which our strategies ignored. Lightweight communication aware strategies may be able to further cut down communication costs in these models.

REFERENCES

[1] R. Fujimoto, "Parallel Discrete Event Simulation," *Comm. of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[2] D. M. Nicol, "The cost of conservative synchronization in parallel discrete event simulations," *J. ACM*.

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

[4] K. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Trans. on Softw. Eng.*, pp. 440–452, September 1979.

[5] R. E. Bryant, "Simulation of packet communication architecture computer systems." MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, Tech. Rep., 1977.

[6] P. M. Dickens, D. M. Nicol, P. F. Reynolds, Jr., and J. M. Duva, "Analysis of bounded time warp and comparison with yawns," *ACM Transactions on Modeling and Computer Simulation*, vol. 6, no. 4, pp. 297–320, Oct. 1996.

[7] M. Schordan, D. Jefferson, P. Barnes, T. Oppelstrup, and D. Quinlan, "Reverse code generation for parallel discrete event simulation," in *Reversible Computation*, J. Krivine and J.-B. Stefani, Eds. Cham: Springer International Publishing, 2015, pp. 95–110.

[8] S. R. Das, R. Fujimoto, K. S. Panesar, D. Allison, and M. Hybinette, "GTW: a time warp system for shared memory multiprocessors," in *Winter Simulation Conference*, 1994, pp. 1332–1339.

[9] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: A high-performance, low-memory, modular Time Warp system," *Journal of Parallel and Distributed Computing*, vol. 62, no. 11, pp. 1648–1669, 2002.

[10] S. J. Turner and M. Q. Xu, *Performance evaluation of the bounded Time Warp algorithm.* University of Exeter, Department of Computer Science, 1990.

[11] 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*, January 1991, pp. 95–103.

[12] J. S. Steinman, "Breathing Time Warp," in *Proceedings of the seventh workshop on Parallel and distributed simulation.* ACM Press, 1993, pp. 109–118.

[13] Z. X. F. Gomes, B. Unger, and J. Cleary, "A fast asynchronous gvt algorithm for shared memory multiprocessor architectures," *SIGSIM Simul. Dig.*, vol. 25, no. 1, pp. 203–208, July 1995. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/214283.214350

[14] R. M. Fujimoto and M. Hybinette, "Computing global virtual time in shared-memory multiprocessors," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 4, pp. 425–446, Oct. 1997. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/268403.268404

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

[16] 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*, ser. PADS '05. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: http://dx.doi.org/10.1109/PADS.2005.27 pp. 39–48.

[17] S. Srinivasan and P. F. Reynolds, Jr., "Non-interfering gvt computation via asynchronous global reductions," in *Proceedings of the 25th Conference on Winter Simulation*, ser. WSC '93. New York, NY, USA: ACM, 1993. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/256563.256834 pp. 740–749.

[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*, ser. PADS '95. Washington, DC, USA: IEEE Computer Society, 1995. [Online]. Available: http://dx.doi.org.proxy2.library.illinois.edu/10.1145/214282.214324 pp. 139–148.

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

[20] 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.*, vol. 24, no. 3, pp. 16:1–16:25, June 2014. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/2611561

[21] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *Journal of parallel and distributed computing*, vol. 7, no. 2, pp. 279–301, 1989.

[22] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, September 1993, pp. 979–993.

[23] L. V. Kale, M. Bhandarkar, and R. Brunner, "Run-time Support for Adaptive Load Balancing," in *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, J. Rolim, Ed., vol. 1800, March 2000, pp. 1152–1159.

[24] H. Menon, N. Jain, G. Zheng, and L. V. Kalé, "Automated load balancing invocation based on application characteristics," in *IEEE Cluster 12*, Beijing, China, September 2012.

[25] G. Zheng, M. S. Breitenfeld, H. Govind, P. Geubelle, and L. V. Kale, "Automatic dynamic load balancing for a crack propagation application," Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. 06-08, June 2006.

[26] A. Bhatelé, L. V. Kalé, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *23rd ACM International Conference on Supercomputing*, 2009.

[27] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, A. Fazenda, C. L. Mendes, and L. V. Kale, "A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model," in *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Itaipava, Brazil, 2010.

[28] C. D. Carothers and R. M. Fujimoto, "Efficient execution of time warp programs on heterogeneous, now platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 3, pp. 299–317, Mar. 2000. [Online]. Available: http://dx.doi.org/10.1109/71.841745

[29] D. W. Glazer and C. Tropper, "On process migration and load balancing in time warp," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 3, pp. 318–327, Mar 1993.

[30] 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.*, vol. 29, no. 9, pp. 1368–1380, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2010.2049044

[31] 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*, May 1998, pp. 46–53.

[32] 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*, ser. SIGSIM PADS '16. ACM, May 2016.

[33] 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*, ser. PADS '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/PADS.2009.21 pp. 35–44.

[34] 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*, ser. SIGSIM-PADS, New York, NY, USA, 2013. [Online]. Available: http://doi.acm.org/10.1145/2486092.2486134 pp. 327–336.

[35] 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*, Dec 2010, pp. 678–687.

[36] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," *ACM Trans. Model. Comput. Simul.*, vol. 9, no. 3, pp. 224–253, July 1999. [Online]. Available: http://doi.acm.org/10.1145/347823.347828

[37] 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*, ser. WSC '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2693848.2694239 pp. 3107–3118.

[38] 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*, ser. SIGSIM-PADS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2901378.2901389 pp. 189–199.

[39] N. Liu, C. Carothers, J. Cope, P. Carns, R. Ross, A. Crume, and C. Maltzahn, "Modeling a leadership-scale storage system," in *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*, ser. PPAM'11. Berlin, Heidelberg: Springer-Verlag, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31464-3_2 pp. 10–19.

[40] 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:*, Nov 2012, pp. 366–376.

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

[42] 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*, ser. SimuTools '13. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013. [Online]. Available: http://dl.acm.org.proxy2.library.illinois.edu/citation.cfm?id=2512734.2512735 pp. 1–9.

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

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

[45] J. J. Galvez, N. Jain, and L. V. Kale, "Automatic topology mapping of diverse large-scale parallel applications," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079104 pp. 17:1–17:10.

[46] N. Jain, "Optimization of communication intensive applications on HPC networks," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2016.

[47] 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," ser. SC, 2014.

[48] E. Deelman and B. K. Szymanski, "Dynamic load balancing in parallel discrete event simulation for spatially explicit problems," in *Workshop on Parallel and Distributed Simulation*, 1998. [Online]. Available: citeseer.nj.nec.com/deelman98dynamic.html pp. 46–53.

[49] M. Choe and C. Tropper, "On learning algorithms and balancing loads in time warp," in *Workshop on Parallel and Distributed Simulation*, 1999. [Online]. Available: citeseer.nj.nec.com/choe99learning.html pp. 101–108.

[50] X. Ni, "Mitigation of failures in high performance computing via runtime techniques," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2016.

[51] G. Zheng, X. Ni, and L. V. Kale, "A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale," in *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[52] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages," in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP '14, Minneapolis, MN, September 2014.

[53] H. Menon, "Meta-balancer: Automated load balancing based on application behavior," M.S. thesis, Dept. of Computer Science, University of Illinois, 2012, http://charm.cs.uiuc.edu/newPapers/12-41.

[54] Y. Sun, J. Lifflander, and L. V. Kale, "PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications," in *Proceedings of 4th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2014*, Munich, Germany, June 2014.

[55] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns, "A case study in using massively parallel simulation for extreme-scale torus network codesign," in *SIGSIM PADS'14*, 2014, pp. 27–38.

[56] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.

[57] I.-H. Chung, R. E. Walkup, H.-F. Wen, and H. Yu, "MPI tools and performance studies—MPI performance analysis tools on Blue Gene/L," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 123.

[58] 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*, Dec 2011, pp. 1–11.

[59] T. L. Wilmarth, "Pose: Scalable general-purpose parallel discrete event simulation," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

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

[61] G. A. Koenig and L. V. Kale, "Optimizing distributed application performance using dynamic grid topology-aware load balancing," in *21st IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[62] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers," in *Proceedings of the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, San Diego, California, USA, September 2010.

[63] O. Sarood, P. Miller, E. Totoni, and L. V. Kale, "'Cool' Load Balancing for High Performance Computing Data Centers," in *IEEE Transactions on Computer - SI (Energy Efficient Computing)*, September 2012.

[64] A. Gupta, O. Sarood, L. Kale, and D. Milojicic, "Improving hpc application performance in cloud through dynamic load balancing," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, 2013, pp. 402–409.

[65] A. Langer, "An Optimal Distributed Load Balancing Algorithm for Homogeneous Work Units," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2597652.2600108 pp. 165–165.

[66] B. Acun and L. V. Kale, "Mitigating processor variation through dynamic load balancing," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2016, pp. 1073–1076.

[67] S. Krishnan and L. V. Kale, "Automating Runtime Optimizations for Load Balancing in Irregular Problems," in *Proc. Conf. on Parallel and Distributed Processing Technology and Applications*, San Jose, California, August 1996.

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

[69] 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*, ser. SC '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503284 pp. 15:1–15:11.