Controlled Asynchronous GVT: Accelerating Parallel Discrete Event Simulation on Many-Core Clusters

Ali Eker
Binghamton University
Binghamton, USA
aeker1@binghamton.edu

Barry Williams
Binghamton University
Binghamton, USA
bwilli33@binghamton.edu

Kenneth Chiu
Binghamton University
Binghamton, USA
kchiu@binghamton.edu

Dmitry Ponomarev
Binghamton University
Binghamton, USA
dponomar@binghamton.edu

ABSTRACT

In this paper, we investigate the performance of Parallel Discrete Event Simulation (PDES) on a cluster of many-core Intel KNL processors. Specifically, we analyze the impact of different Global Virtual Time (GVT) algorithms in this environment and contribute three significant results. First, we show that it is essential to isolate the thread performing MPI communications from the task of processing simulation events, otherwise the simulation is significantly imbalanced and performs poorly. This applies to both synchronous and asynchronous GVT algorithms. Second, we demonstrate that synchronous GVT algorithm based on barrier synchronization is a better choice for communication-dominated models, while asynchronous GVT based on Mattern's algorithm performs better for computation-dominated scenarios. Third, we propose Controlled Asynchronous GVT (CA-GVT) algorithm that selectively adds synchronization to Mattern-style GVT based on simulation conditions. We demonstrate that CA-GVT outperforms both barrier and Mattern's GVT and achieves about 8% performance improvement on mixed computation-communication models. This is a reasonable improvement for a simple modification to a GVT algorithm.

KEYWORDS

Parallel Discrete Event Simulation, Intel Xeon Phi, Knights Landing, Manycore Architectures, Performance, Global Virtual Time

ACM Reference Format:
Ali Eker, Barry Williams, Kenneth Chiu, and Dmitry Ponomarev. 2019. Controlled Asynchronous GVT: Accelerating Parallel Discrete Event Simulation on Many-Core Clusters. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337927

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337927

1 INTRODUCTION

The recent proliferation of many-core processors offers the promise of achieving scalable high-performance parallel discrete event simulation (PDES) on commodity systems. The emergence of Intel's 64-core Xeon Phi processors was recently followed by the announcements from both Intel and AMD featuring the increase in the number of cores of their mainstream processors to 48 (Intel 48-core Cascade Lake Xeon) and 64 (AMD EPYC Rome) respectively. With these many-core systems soon becoming common place, recent research efforts examined performance aspects of PDES in these environments [7, 9, 17, 31, 33]. However, these previous efforts were confined to a single-node system.

In this paper, we extend these studies to clusters of many-core CPUs, and in particular investigate the impact of Global Virtual Time (GVT) algorithms on PDES performance in these environments. GVT algorithms in optimistic simulation typically come in two flavors: 1) synchronous GVT, where simulation threads use barrier synchronization at GVT intervals; and 2) asynchronous GVT, where threads continue simulation while GVT computation rounds occur in the background. The asynchronous algorithms are rooted in Mattern's algorithm for distributed environments [23]. Recent work proposed further optimizations, such as wait-free GVT [25], but those are designed for shared memory architectures. Since this paper targets distributed cluster environment, we consider a slightly modified Mattern's algorithm as our asynchronous GVT.

The results of this study lead to several key conclusions. First, we establish that when MPI-based cross-node communication is part of simulation, this communication must be handled efficiently. When every simulation thread is also processing MPI messages, performance degrades rapidly even for small percentage of remotely generated events, because threaded MPI performance is inherently limited by the lock contention among threads, as is well established in high-performance community [2]. To address these locking limitations, a solution developed in [31] dedicates a single thread within each core to handle all MPI messages, both for sending and receiving. In the original work of [31], this MPI thread is also tasked with normal event processing, similar to other threads. The study of [31] was performed on nodes with small number of cores, therefore dedicating one of those cores entirely to MPI messages may constrain the event processing capabilities of the node significantly. However, we observed that when the MPI thread is also performing

event processing, the events processed by this thread are delayed, creating a virtual time disparity between the simulation progress among threads and causing significant increase in rollbacks and drop in simulation efficiency. For many-core systems with an abundance of cores available on each node, we propose to relieve MPI threads of the responsibility to handle simulation events, i.e., these threads should be exclusively dedicated to MPI processing - one for each node. We show that such a model with an isolated MPI thread significantly outperforms the model where MPI threads are also doing event processing. This is true for both synchronous and asynchronous GVT algorithms, and the disparity increases with the number of simulation threads. This is the first contribution of this paper.

Our second observation relates to the comparison of synchronous and asynchronous GVT algorithms, assuming a model with a dedicated MPI thread at each node. We show that asynchronous GVT based on Mattern's algorithm performs better for models that are dominated by computation, rather than communication. We control this ratio by varying the event processing granularity (EPG), the computational cost of processing an event. Computation-dominated models typically result in high simulation efficiency and smaller number of rollbacks, thus making the asynchronous style thread progress more beneficial without suffering the adverse effects of rollbacks. At the same time, for communication-dominated models, we demonstrate that a synchronous barrier-based GVT algorithm often produces better performance, as complete thread synchronization at GVT intervals limits the disparity in threads' virtual time and reduces the amount of rollbacks. We also perform these comparisons with imbalanced models, and observe similar results.

Motivated by these observations, we propose a modification to asynchronous GVT that imposes additional synchronization when it is determined to be beneficial for performance. This scheme, which we call Controlled Asynchronous GVT (CA-GVT), performs close to synchronous GVT in communication-driven scenarios, and close to asynchronous GVT in computation-driven scenarios. The decision to impose additional synchronization in CA-GVT is driven by the observation of the simulation efficiency. The efficiency readings below a threshold (due to high rollbacks) trigger synchronization. We show that for models that have a mix of computation and communication phases, CA-GVT shows a robust 7-8% performance improvement compared to either a synchronous or asynchronous GVT. This is a reasonable gain for a simple modification to a GVT algorithm.

The rest of the paper is organized as follows. Section 2 overviews our evaluation and experimentation methodology, including ROSS simulator and the details of our hardware architecture. Section 3 reviews the GVT algorithms considered for this study. Section 4 presents the results of using existing GVT algorithms and compares their performance under different simulation models. Section 5 describes CA-GVT algorithm and Section 6 presents its results. Section 7 reviews the related work and we conclude in Section 8.

2 BACKGROUND AND EXPERIMENTAL SETUP

In this section, we review PDES basics and describe our evaluation methodology and infrastructure.

Overview of PDES

A discrete event simulation (DES) consists of multiple logical processes (LP), where each LP represents a physical entity in the simulated system. Such a simulation may be distributed over multiple cores or nodes, in a parallel discrete event simulation (PDES). LPs communicate via time stamped event messages to simulate the interactions between physical entities of the real system [10, 19]. LPs advance in the virtual simulation time by processing the time stamped events received from other LPs. Each event processing leads to generating and sending a new event for any of the other LPs. LPs maintain a *Local Virtual Time (LVT)* to indicate their virtual position in the simulation. The time stamp of the last processed event determines the LVT of an LP.

Event processing should preserve the causality order between event messages based on their time stamp. For instance, two consecutive event messages sent by an LP should be processed by the receiver LP in the same order. Approaches to ensure this can broadly be divided into two categories: (1) a conservative approach where a global synchronization of LPs and acknowledgment messages are used to guarantee that out-of-order event processing never occurs, and (2) an optimistic concurrency based approach. In the latter, each LP executes events in received order without checking the causality order. Therefore, there is a possibility for processing an event with a time stamp less than the processing LP's LVT. Such events violate the causality order and are called *straggler events*. Straggler events cause an LP to roll its state back to a point in virtual time to just before the straggler's virtual time. During the rollback, an LP reverts all the messages it sent optimistically and then process the straggler event. After the rollback, an LP can safely process events in the order of their time stamp. Optimistic simulation allows straggler events and maintains the causality order by rolling back the LPs to a state before the straggler event, and then replaying the simulation forward.

Rollbacks in an optimistic simulation can be performed by either checkpointing the entire LP state, or by using reverse computation. In either case, some state or event history must be maintained in order to administer rollback mechanism. Event histories grow over time as LPs generate more messages. Without some way to trim these histories, eventually all memory would be consumed. This trimming is called *fossil collection*, and relies on computing a time T such that it can be guaranteed that no rollbacks to before time T will ever occur in the future. In other words, an LP must not receive a straggler message with a time stamp less than T so that it can free the event histories prior to T. T can be determined by computing the minimum of (1) the minimum LVT among all LPs and (2) the minimum time stamp of all in-transit messages (possibly including straggler messages). This minimum time is called the global virtual time (GVT).

Computing the GVT efficiently is crucial for the scalability of a PDES performance. Because, the frequency and correctness of a GVT computation determines the memory footprint and controls the synchrony between LPs. GVT is also needed to commit irreversible operations and to keep track of the actual simulation progress. GVT algorithms and their implementation details are discussed further in Section 3.

There are three types of event messages with respect to their destination. These are local, regional and remote messages. Local messages are the messages sent by an LP to itself thus they do not require access to the other cores. They do not traverse the interconnect and have the fastest transmission time. Second type is the regional messages where the destination is one of the cores residing in the same node as the sender core. Sender and receiver communicate through the shared memory, thus a locking mechanism is needed. Slowest message type is the remote messages where the message should be sent to a core in a different node through network.

Experimental Setup and Metrics

We perform our experiments on a 8-node cluster of Intel KNL model 7230 processors [1, 12] connected by a 10 GBit Ethernet network. Experiments were conducted on CentOS 7.2 using mpich-3.3 and mpicc as the MPI and compiler versions. KNL CPUs contain 64 cores, each with four simultaneous hardware threads. The clock frequency can be up to 1.3 GHz. KNL cores also feature branch prediction and out-of-order execution logic. Cores are paired into tiles, each with a 1 MB L2 cache. Our tests were conducted using KNL processors with 96 GB of DDR4 memory and 16 GB on-package fast RAM (L3 cache), called MCDRAM.

For our experiments, we used a modified version of ROSS simulator [5] as our PDES engine. ROSS implements the optimistic approach to preserve the causality order. Our version is modified to be multithreaded rather than multiprocess [18].

PHOLD benchmark is a simplistic but highly configurable model. It initially generates and assigns the same number of starting events per LP. Each LP randomly chooses a destination LP at every simulation iteration and sends messages to it based on the remote message percentage specified. We modified PHOLD to generate various models where simulation is dominated by computation or communication overheads. Specifically, we vary the thread count, the percentage of remotely generated events, and the event processing granularity (EPG). The EPG represents the amount of work required to process a single event, and is specified in units approximately equal to one FLOP per unit. Our goal is to understand the behavior and scaling trends of the ROSS simulator on a cluster of KNL nodes for different GVT algorithms.

We report the performance results in terms of committed event rate and efficiency. Efficiency of the simulation system is calculated by taking the ratio of the committed events (number of events which are not reverted due to a rollback) over total number of events generated in the system. Committed event rate shows the total number of committed events per second. As we increase the number nodes, we maintain the number of starting events per node, thus proportionately increasing the total number of events generated by the simulator. The number of LPs per node and the number of starting events per LP stay the same as we increase the number of nodes.

We assigned 128 LPs for each hardware thread and 1 starting event for each LP. We load the KNL nodes with 60 threads per node (one thread per core, leaving some spare cores for the activities of operating system without disrupting the simulation). Note that because each event generates exactly one other event, there is a fixed number of events in the total simulation at any instant in time.

If the underlying system is capable of efficiently keeping up with the increasing load without incurring additional delays, we can expect the committed event rate to also show improvements commensurate with the increase in the number of nodes. This is known as *weak scaling* [4].

3 GLOBAL VIRTUAL TIME (GVT)

In an optimistic simulator, the state of the LPs must be saved continuously throughout the simulation. This required because conceptually a rollback could target any of the previous states of an LP. Intuitively, these saved states grow larger as simulation progresses, thus creating memory exhaustion and lower cache utilization. In order to prevent this, the Global Virtual Time (GVT) is periodically computed to determine the earliest time a rollback could target so that LPs' states prior to GVT could be freed. GVT is essentially the greatest lower bound on the local virtual time of all LPs and the time stamp of all in-transit messages.

Synchronous GVT

Synchronous GVT algorithms essentially follow the "stop-synchronize-and-go" model where each LP periodically stop processing, synchronize at a barrier, wait until all transient messages arrive and collectively compute the new GVT value. Synchronous implementations may be inefficient when LPs arrive the barrier at different times. The faster LPs that arrive early must wait while slower LPs are catching up. During this idle time, none of the core simulation tasks such as event processing is being accomplished. This cycle repeats at each GVT round.

The synchronous GVT algorithm deployed in our many-core cluster is composed of two levels of synchronization points. First, all the threads in each node synchronize at a pthread barrier to let all intra-node (regional) messages to arrive their destination cores. Second, each simulation instance (one per node) synchronizes at an MPI barrier to let inter-node (remote) messages to arrive their destination nodes. All the LPs in the system are blocked until there is no more in-transit messages between any pair of LPs. In other words, LPs cannot do useful simulation work until there are no more regional or remote messages still in transmission.

A diagram of the barrier GVT computation is shown in Figure 1. Horizontal lines depict the wall-clock time of four LPs and arrows show the event messages sent between LPs. Once the GVT computation begins, all processing elements are blocked until all the messages in the system have been received. The black circles depict the point when an LP calls the pthread barrier and becomes idle. Dashed lines show the time elapsed while an LP is blocked during the idle time in the barrier call. Idle time lasts until all the LPs call the barrier and there is no more in-transit messages in the system. No new event processing or event generation occurs during the dashed lines.

The pseudo-code for the barrier-based GVT computation is shown in Algorithm 1. In this algorithm, LPs first read the messages sent to them (line 3) and then compute the difference between the number of messages they sent and received. This difference is held

in *msgCount* (line 4). The LPs then call pthread barrier and synchronizes in line 5. *PthreadBarrierSum* is a map-reduce operation between shared memory threads which takes *msgCount* of each LP as an input and reduces them into *transitNode* as an output using the sum operation. *transitNode* shows the number of in-transit messages in a node. Then, the LPs which are responsible for the MPI communication between nodes call MPI barrier and synchronize in line 7. At this point other threads wait the MPI thread in line 12 until it completes the MPI operation. *MpiBarrierSum* is a map-reduce operation between nodes which takes *transitNode* of the participating LPs as an input and reduces them into *transitTotal* as an output using the sum operation. *transitTotal* shows the total number of in-transit messages in the system. This process is repeated in a tight loop until *transitTotal* is checked as 0 by each LP in line 8. When the LPs break out of the loop, they synchronize one last time to reduce their LVTs into GVT using min operation. At this point, there are no in-flight messages, thus a new GVT value can be computed as the minimum LVT among all of the LPs. Once all LPs get the new GVT, they fossil collect and event processing continues until the next GVT round.

GVT interval determines the gap between two consecutive GVT rounds. LPs loop over the core simulation cycle the predefined GVT interval times and participate the simulation tasks at each iteration. When the GVT interval counter reaches the GVT interval constant, GVT round is initiated. Thus, GVT interval is based on the number of events processed, not the virtual time elapsed.

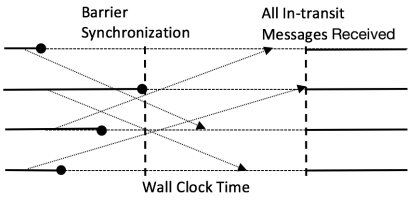

Figure 1: Snapshot of a Barrier GVT Computation Cycle

Algorithm 1 Barrier GVT Algorithm

```
1:  procedure ComputeGVT-Synchronously
2:      while 1 do                          ▷ Loop until all in-transient messages received
3:          ReadMessages()
4:          msgCount = LP.MsgSent − LP.MsgReceived
5:          transitNode = PthreadBarrierSum(msgCount)
6:          if LP is responsible for MPI then
7:              transitTotal = MpiBarrierSum(transitNode)
8:          if transitTotal = 0 then
9:              Break                       ▷ All in-transit messages received
10:     nodeGVT = PthreadBarrierMin(LP.LVT)
11:     if LP is responsible for MPI then
12:         GVT = MpiBarrierMin(nodeGVT)
13:     PE.GVT ← GVT
14:     fossilCollect()
```

Asynchronous GVT

In contrast, asynchronous GVT algorithms proceed "in-line" with the core simulation tasks which yields event receiving, processing and sending to be interleaved with the GVT computation. GVT is computed at the background, asynchronously however, a higher computational overhead may occur due to the management of the LPs' participation in the GVT process. We used Mattern's asynchronous GVT algorithm [23] which is based on circulating a control message among LPs to accumulate the message counts, the minimum LVT and the minimum time stamp as a basis for our GVT implementation.

For this study, we adapted Mattern's distributed GVT algorithm to make it more suitable for a cluster of many-core architectures. Two kinds of control messages are utilized: One is a shared memory structure which is accessed by each LP residing on the same node. This accumulates the in-transit messages in a node. When each LP accumulates its message count at the shared control message, the second kind of control message is spawned. This control message is an MPI data type which circulates in a ring of simulation instances to accumulate in-transit messages between nodes. When inter-node control message finishes its circulation, new GVT can be computed by taking the minimum of *(A)* the minimum LVT among all the LPs, *(B)* the minimum time-stamped non processed event in the system. *A* and *B* are also accumulated at the control message so that each LP has the information to compute the same GVT locally.

Each LP has two phases, white and red. Events are also colored as white and red based on sending LP's color. A GVT round is initiated the same way as in the synchronous algorithm based on the GVT interval. When a GVT round is initiated, each LP transits from their normal white phase to the red phase.

During white phase, each LP counts the number of white messages they send and receive. The difference of these is accumulated into the control message. When the total accumulation is checked as 0, there are no more in-transit white messages in the system. During red phase LPs record the time stamps of the red messages they send. An LP computes the minimum of these time stamps and accumulates it into control message together with its LVT after it checks the white message counter in control message as 0. GVT can be computed by taking the minimum of *(A)* minimum LVT and *(B)* minimum time stamped red message accumulated in the control message.

A timing diagram of the asynchronous algorithm is shown in Figure 2. Horizontal lines show the wall-clock time line of four LPs. For clarity, assume that all LPs change their phases and check control message in order (this assumption is not necessary in practice, but it simplifies the explanation). Messages are shown as arrows. The sending (+1) and receiving (-1) white messages are counted locally by each LP as shown.

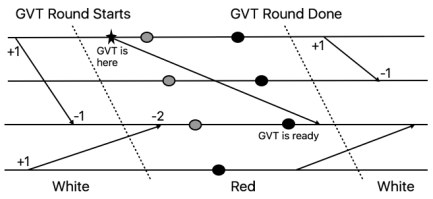

Figure 2: Snapshot of a Mattern's GVT computation

After the transition to the red phase (shown as the first tilted dashed line), the message counts start to be accumulated at the

control message. The first LP which accesses the control message sees it as 1. This is shown in the form of a grey circle on the first line. Then, the second LP accesses the control message and it also checks it as 1 since it has no event counts to accumulate. Then, the third LP with the message count of -2 accesses the control message and updates it from +1 to -1. This is depicted as another grey circle, implying that there are still in-transit messages in the system. Finally, the fourth LP arrives and accumulates its +1 event count with the control message and checks it as 0. At this point there are no in-transit white messages in the system so LPs can start accumulating their LVT and minimum time-stamped red message. This is shown as a black circle at the bottom line.

The LPs accumulate their minimum red message timestamps and LVTs into the control structure as they pass the black circles. The last LP that reaches the black circle computes the GVT by taking the minimum of *A* and *B*. At this point, the control message holds the LVT of the fourth LP since it has the smallest timestamp. However, the red event from the first LP has an even smaller timestamp, which possibly makes it a transient event. Therefore, the GVT is set to the timestamp of that event. After that, all LPs read this new GVT value, turn their color into white, and start counting events again. The pseudo-code for this asynchronous GVT algorithm is shown in Algorithm 2.

Algorithm 2 Mattern's GVT Algorithm

```
1:  procedure ComputeGVT-Asynchronously
2:      if LP = white then
3:          LP ← Red
4:          LP.min_red ← ∞
5:          accumulateMsgCountersInNode()
6:          if LP is responsible for MPI and All LPs are red then
7:              accumulateMsgCountersAcrossNodes()
8:      else                                                    ▷ LP is red
9:          if All white messages are received then
10:                                                  ▷ CM is Control Message
11:             CM.LVT ← min(CM.LVT, LP.LVT)
12:             CM.min_red ← min(CM.min_red, LP.min_red)
13:         if LP is responsible for MPI and LPs checked CM then
14:                                  ▷ All LPs checked white message counter as 0
15:             circulateGlobalCM()
16:         if CM done circulation then
17:             LP ← White
18:             GVT ← min(CM.LVT, CM.min_red)
19:             fossilCollect()
20:             resetCM()
```

LPs start the simulation as white and turn to red as shown in line 3 in Algorithm 2. Once an LP is red, it accumulates its message count into the shared memory control message using *accumulateMsgCountersInNode* routine (line 5). When each LP has done that, the message counters across nodes are accumulated by the LP responsible for MPI using *ccumulateMsgCountersAcrossNodes()* routine (line 7). After that point, LPs check the control message until all the white messages are received (line 9). Unlike the synchronous GVT, LPs are not blocked and participate the core simulation tasks while computing the GVT at the background. Once an LP checks the number of in-transit white messages as 0, it updates the control message's LVT and minimum time stamped red message (*min_red*) as shown in lines 11 and 12. Control message is circulated one last time to accumulate every LP's LVT and *min_red* using *circulateGlobalCM()* (line 14). After the circulation is done, LPs turn back to white and GVT can be computed as the control message's LVT and *min_red* (lines 16-17). Finally, LPs fossil collect, reset the control message and keep counting their white messages until the next GVT round.

4 EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we present the results of our experiments in two parts. First, we show the effects of offloading the MPI communication from event processing for communication and computation-dominated scenarios. Then, we evaluate Barrier (synchronous) and Mattern's (asynchronous) GVT algorithms under the computation and communication-dominated scenarios. All the experiments are performed on a cluster of eight KNL nodes each running sixty threads. We limited our evaluations to one thread per core, as overloading cores by executing multiple threads on them was shown to create contention and slow down simulations [33]. This is especially true in a cluster environment.

We modified the classical PHOLD benchmark [11] to create various simulation models. Specifically, for the computation-dominated scenario, we used 10% regional messages, 1% remote messages and 10K EPG, and for the communication-dominated scenarios we assumed 90% regional messages, 10% remote messages and 5K EPG. All graphs show the committed event rate of ROSS simulator on the y-axis and the number of KNL nodes on the x-axis.

The GVT interval of 50 is chosen for the experiments in the next subsection and 25 is chosen for computation and communication-dominated scenerios. These numbers are chosen because they resulted in the best overall performance.

Dedicated MPI Thread

In order to alleviate the scalability issues created by MPI threads that also perform event processing, we first propose to isolate all MPI calls to a dedicated thread, which does not perform normal event processing. Remaining threads are called *worker threads* and they perform basic simulation functions such as event processing, sending and receiving of the event messages.

Worker threads write their remote messages into a global shared data structure to be read by the MPI thread which sends them over the network. Similarly, worker threads receive remote messages by reading another global shared data structure which is populated by the MPI thread upon receiving an MPI message.

The dedicated MPI thread is also responsible for MPI tasks involved in GVT functions. In Mattern's algorithm, the control message is circulated between nodes using point-to-point MPI communication routines. In Barrier GVT algorithm, MPI collective communication routines are utilized for synchronization and reduction operations as explained in Section 3. There is only one dedicated MPI thread per node which manages all MPI communications in ROSS.

As seen from the results, using a dedicated MPI thread to isolate MPI communication from event processing is critical for scalability and it improves performance of both GVT algorithms at every node count. For example, in Figure 3 which presents a computation-dominated scenario, Mattern and Barrier GVT algorithms with a dedicated MPI thread outperform their standard implementations by 51% and 17% respectively when the node count is eight. Similarly, in a communication-dominated scenario presented in Figure 4,

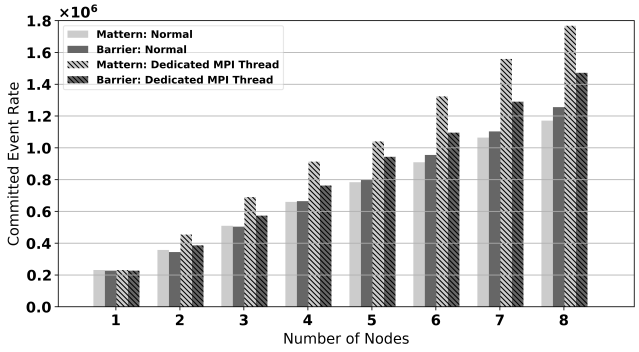

Figure 3: Dedicated MPI Thread for Computation-Dominated Workload

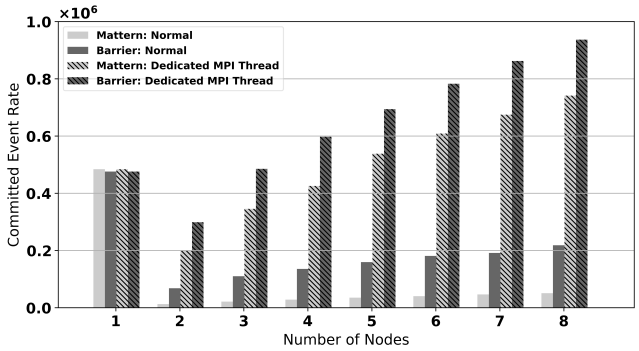

Figure 4: Dedicated MPI Thread for Communication-Dominated Workload

performance of Mattern and Barrier GVT increases 14.59X and 4.29X respectively.

Performance benefits of the dedicated MPI thread increase significantly in a communication-dominated scenario, since MPI communication impacts the performance more in a communication-dominated case compared to a computation-dominated and the dedicated MPI thread plays a more important role in the simulation.

A dedicated MPI thread helps to improve performance of both GVT algorithms for different reasons. The asynchronous nature of Mattern's algorithm allows any LP with a higher workload to fall behind in time, thus creating disparity between LPs and impacting simulation efficiency. Without a dedicated MPI thread, at least one of the worker threads also needs to assume MPI functions, thus becoming a bottleneck. This thread then sends regional or remote messages which have a higher chance to cause a rollback. Offloading the MPI communication to a dedicated thread reduces this disparity between LPs and increases efficiency. Efficiency of Mattern's GVT algorithm increases from 88.75% to 92.11% using eight nodes.

On the other hand, Barrier GVT algorithm's main overhead is idle threads blocked at the barrier during synchronization. Faster LPs which arrive at the barrier earlier waste more time waiting for slower LPs. Efficiency is not impacted significantly, because threads are synchronized and the MPI thread with higher workload is not allowed to fall behind. For example, the efficiency of Barrier GVT decreases from 91.53% to 91.17% with a dedicated MPI thread on eight nodes. Performance is still improved by a dedicated MPI thread because remote messages are sent more efficiently. Intuitively, when the MPI thread does not need to process, send or receive messages, it performs MPI functions more frequently thus resulting in faster inter-node communication and faster simulation. For example, the normal Barrier implementation results in 16.43 seconds wall-clock run time for simulation to complete while the dedicated MPI thread results in 11.53 seconds although two implementations' committed event numbers are same.

In the following subsections, we only present the results of simulations with a dedicated MPI thread. In the next subsection we present the results for a computation-dominated scenario and analyze why Mattern's GVT algorithm performs better than Barrier GVT.

Computation-Dominated Scenario

We created a computation-dominated scenario by using a coarse event processing granularity to tilt the simulation from communication overheads towards event processing. As seen from Figure 5, *Mattern's asynchronous GVT algorithm performs better than Barrier GVT implementation when computation dominates over communication.* Mattern's GVT is 27.9% faster than Barrier GVT on eight nodes.

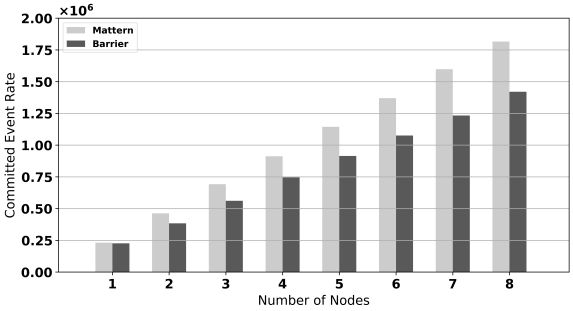

Figure 5: Performance Comparison of Mattern and Barrier for Computation-Dominated Workloads

This performance trend in cases with high event processing granularity is due to asynchronous nature of Mattern's algorithm. Mattern's GVT is faster since it does not require any synchronization and allows LPs to advance optimistically. The periodic stopping of Barrier GVT detrimentally impacts the performance since, with high EPG, threads may be blocked for a proportionally longer period of time at the barrier while waiting for other threads to complete. For example, when the EPG value is increased from 10K to 40K the time elapsed on the Barrier GVT function increases from 9.52 seconds to 12.08 seconds.

Communication-Dominated Scenario

In this section, we created communication-dominated scenarios by setting high remote and regional message percentages and low

event processing granularity. LPs spend more time for message transmission than event processing thus simulation suffers from a communication load rather than the EPG delay. *We observe that the inter node MPI communication is a major bottleneck for scalability when simulation model is dominated by communication over computation.* Fine event processing granularity creates a communication-dominated scenario where remote events produce more rollbacks, lower efficiency and lower event commit rate.

We observe that the heavier communication load degrades the overall performance more than the computation delays for both algorithms. For example, when communication dominates over computation, performance decreases by 44.3% and 18.4% for Mattern and Barrier GVT algorithms respectively. The performance drop of Matter's GVT can be credited to inter-node communication bottleneck. The performance degradation expends as slower communication causes more rollbacks, especially with Mattern's GVT algorithm. For example, the number of rollbacks increases 6.4X from 2646768 to 16867624 and the efficiency decreases from 92.08% to 64.24% when communication dominates over computation using eight nodes. Because, the work load on the MPI thread grows which further increases the impact of the MPI bottleneck.

For Barrier GVT algorithm, the main reason of performance drop is not the inefficiency but a slower simulation progress. For example wall-clock run time of the simulation increases from 21.05 seconds to 25.64 seconds when communication dominates over computation. Because the LPs wait longer at barrier calls to allow heavier message load to arrive its destination. The time elapsed on the Barrier GVT function increases from 8.92 seconds to 31.38 seconds.

As seen from Figure 6, Barrier GVT outperforms Matter's GVT algorithm by 14.5% using eight nodes. The periodic synchronization of Barrier GVT reduces the possibility of a straggler message because LPs line up at every GVT round and wait for all the in-transit messages to arrive. On the other hand, Mattern's GVT requires no synchronization so LPs have more chances to fall behind because of the heavier communication load of the communication-dominated model. For example, the efficiency of Barrier GVT is 94.2% while it is 64.3% for Mattern's GVT algorithm. Also, the virtual time disparity between LPs widens for Mattern's GVT algorithms when communication dominates over computation. We computed the disparity by calculating the standard deviation among LVTs at each GVT round. We then summed each round's disparity and divided it by the number of GVT rounds to compute an average standard deviation of the simulation system. This number is 0.31 for Barrier GVT while it is 0.43 for Mattern's GVT algorithm. As a result, we conclude that *when communication dominates over event processing, Barrier GVT outperforms Matter's GVT algorithm significantly.*

5 CONTROLLED ASYNCHRONOUS GVT

In this subsection, we present Controlled Asynchronous GVT (CA-GVT), a new GVT algorithm that adaptively adds synchrony to Mattern's algorithm based on the simulation progress to avoid performance problems associated with rollbacks. Specifically, we periodically keep track of the number of rollbacks over total event messages, and if this number exceeds a predetermined threshold, the LPs synchronize during the GVT round. As shown by Algorithm 3, three synchronization points are added: one to the white phase of Mattern's algorithm (line 4), one to the red phase (line 14), and one after the fossil collection (line 30). At the end of each round, efficiency is computed based on the events committed so far (line 31) and next round's GVT is either computed asynchronously or synchronously.

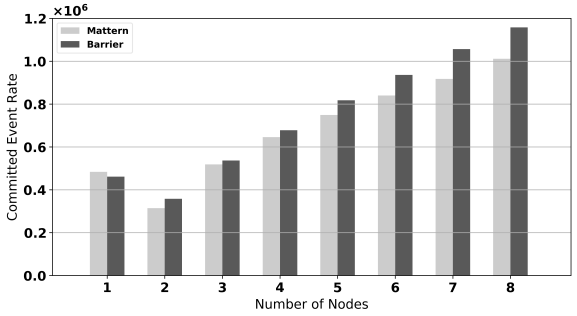

Figure 6: Performance Comparison of Mattern and Barrier for the Communication-Dominated Scenario

CA-GVT performs asynchronously when efficiency is high and allows more optimistic processing (without barrier stalls). On the other hand, CA-GVT performs synchronously when efficiency is low due to the virtual time disparity between LPs, which happens in communication-dominated scenarios with significant percentage of remote events. This synchronization increases efficiency as it aligns the progress of LPs, thus resulting in significant performance increase compared to pure asynchronous GVT implementation under a communication-dominated model.

Algorithm 3 Controlled Asynchronous GVT Algorithm

```
1:  procedure ComputeGVT
2:      if LP = white then
3:          if SyncFlag = True then
4:              barrier()
5:          LP ← Red
6:          LP.min_red ← ∞
7:          accumulateMsgCountersInNode()
8:          if LP is responsible for MPI and All LPs are red then
9:              accumulateMsgCountersAcrossNodes()
10:     else                                                  ▷ LP is red
11:         if All white messages are received then
12:                                                ▷ CM is Control Message
13:             if SyncFlag = True then
14:                 barrier()
15:             CM.LVT ← min(CM.LVT, LP.LVT)
16:             CM.min_red ← min(CM.min_red, LP.min_red)
17:         if LP is responsible for MPI and LPs checked CM then
18:                                  ▷ All LPs checked white message counter as 0
19:             circulateGlobalCM()
20:         if CM done circulation then
21:             if Efficiency < Threshold then
22:                 SyncFlag ← True
23:             else
24:                 SyncFlag ← False
25:             LP ← White
26:             GVT ← min(CM.LVT, CM.min_red)
27:             fossilCollect()
28:             resetCM()
29:             if SyncFlag = True then
30:                 barrier()
31:             computeEfficiency()
```

Figure 7 shows the timing diagram of CA-GVT. Similar to Mattern's algorithm, CA-GVT algorithm requires LPs to keep track of messages sent and received in the white phase. In the red phase, LPs accumulate their message counters in the control message to check if the number of messages they received is equal to the number of messages sent to them. The grey circles depict the time an LP checks the control message. The difference from a pure asynchronous implementation is that LPs align at red and white phases so that none of the messages are allowed to pass to the next phase. GVT is computed by taking the minimum of two variables: minimum LVT and minimum time-stamped red messages. A new GVT value becomes ready to be computed when the last LP checks the control message.

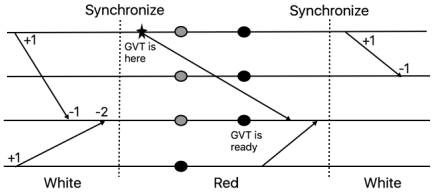

Figure 7: Snapshot of CA-GVT Computation

6 CA-GVT RESULTS

In this section, we compare and analyze the performance of CA-GVT with pure synchronous Barrier and pure asynchronous Mattern's GVT algorithms. We first evaluate them on a computation and communication-dominated scenarios in Figure 8 and Figure 9 respectively. We then experiment with several mixed models where the simulation phases alternate between computation and communication dominated scenarios. Figure 10 shows the committed event rates for a model where simulation runs for the first 10% of its execution time under computation-dominated workload, then spends the next 15% under communication-dominated workload, and the pattern repeats. We refer to this mixed model as 10-15 model. In general, the notation X-Y refers to a model where the first X% of simulation time is spent in computation-dominated mode, and the following Y% spent in communication-dominated mode, with the pattern repeated. Figure 11 and Figure 12 show the results for 15-10 and 5-5 mixed models respectively.

As previously explained, Mattern's GVT performs the best under the computation-dominated scenerio. CA-GVT performs 8% slower than Mattern and 19% faster than Barrier GVT algorithms using eight KNL nodes. CA-GVT detects the higher efficiency and executes at asynchronous mode to take advantage of the computation-dominated model. It slightly under performs Mattern's algorithm because of the extra overhead of the efficiency computation at each GVT round which results in slower GVT computations. For example, the average CPU time spent during a GVT round is 4.4 seconds for Mattern's GVT while this number is 4.78 for CA-GVT. This time includes event processing too since Mattern's GVT computation is interleaved with other simulation tasks. CA-GVT's efficiency threshold for switching to the synchronous mode is 80% and simulation is run with 92.98% efficiency, thus CA-GVT executes at

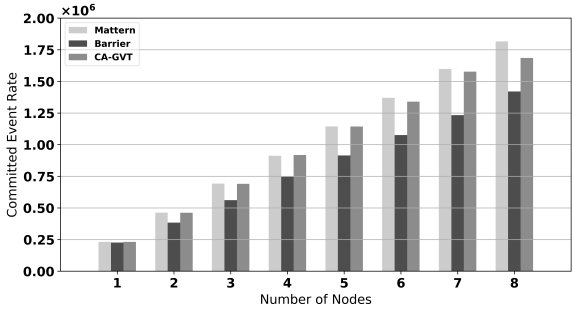

Figure 8: Performance Comparison of Mattern, Barrier and CA-GVT for Computation-Dominated Workloads

asynchronous mode constantly for the computation-dominated scenario.

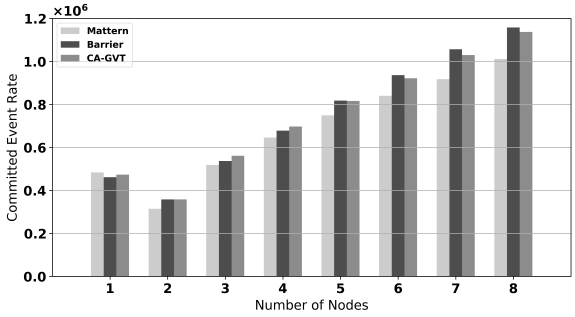

Figure 9: Performance Comparison of Mattern, Barrier and CA-GVT for Communication-Dominated Workload

Barrier GVT performs the best under communication-dominated scenarios. CA-GVT performs 2% slower than Barrier and 13% faster than Mattern's GVT algorithms using eight KNL nodes. Similarly, CA-GVT detects the lower efficiency related to heavier communication and switches to synchronous mode. Specifically, it detects the lower efficiency in the first GVT round, switches to the synchronous mode and executes the next 157 GVT rounds synchronously. At this point, efficiency reaches the threshold, CA-GVT switches back to asynchronous mode and executes 168 of the next 178 GVT rounds asynchronously. Simulation completes with 79.95% efficiency which is driven by the CA-GVT's efficiency threshold. On the other hand, Mattern and Barrier GVT complete simulation with 36.19% and 85.32% efficiency respectively.

CA-GVT algorithm outperforms Mattern/Barrier GVT algorithms by 8.3%/6.4%, 6.9%/12.7% and 7.8%/8.3% under 10-15, 15-10 and 5-5 mixed models respectively, using eight nodes. As seen from the graphs, CA-GVT can adapt itself to the simulation model and benefits from synchronous and asynchronous GVT algorithms to execute in optimum synchrony when computation and communication are interleaved. The percentage of the simulation executed synchronously by CA-GVT is dependent on the efficiency threshold.

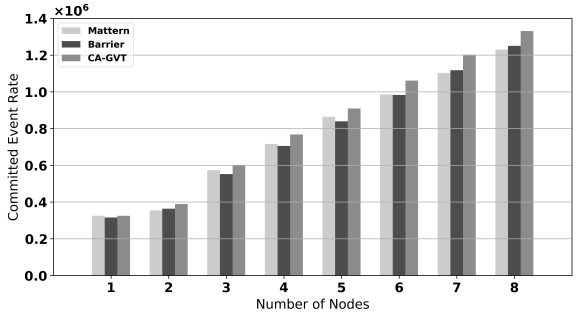

Figure 10: Performance for 10-15 Mixed Model

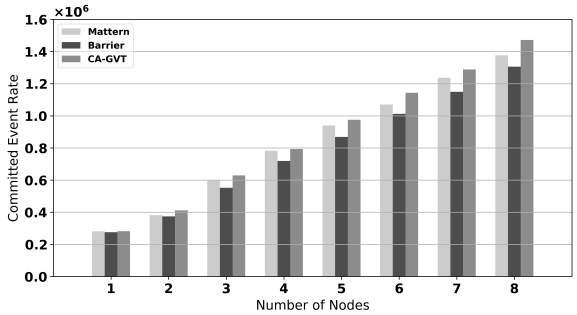

Figure 11: Performance for 15-10 Mixed Model

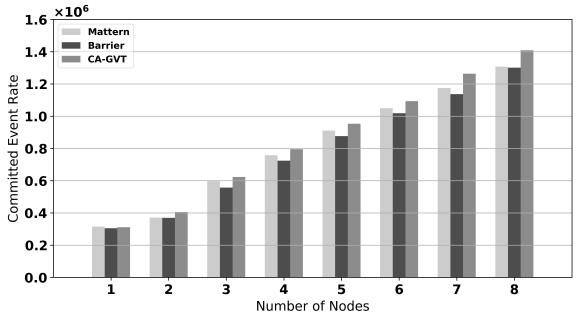

Figure 12: Performance for 5-5 Mixed Model

7 RELATED WORK

GVT computation has been studied extensively in the literature, though primarily in a distributed setting. Samadi [28] developed one of the first GVT algorithms and introduced the transient message and simultaneous reporting problem. That algorithm, however, requires that acknowledgement messages be sent, causing extra communication overhead. Chandy and Lamport [6] describe one of the first distributed snapshot algorithms. Mattern [23] built on that to develop an asynchronous algorithm that does not require acknowledgement messages.

There has also been work to improve the performance of GVT on multiple cores. Eker et al. [9] evaluated the impact of synchronous and asynchronous GVT algorithms on PDES performance on single-node KNL systems. The main conclusions were that synchronous GVT is a better choice for imbalanced models, while asynchronous GVT provides better performance point for balanced models. In this paper, we extend these studies to clusters of KNL processors and evaluate GVT algorithms that work with network-based communication using MPI. We show that the choice of optimal GVT algorithm is determined by the communication/computation balance, and we propose a flexible GVT algorithm that performs well under all operating conditions.

The work by Ianni [16] develops a non-blocking algorithm for concurrent computation of GVT. In [20], the researchers developed an asynchronous algorithm for computation of GVT. In [8], the authors developed a multicore GVT based on Samadi's algorithm for a simulator written in the Go language. The works of [29, 30] are early studies on controlled synchrony of GVT computations using SPEEDES framework.

There has also been significant work investigating PDES on manycore architectures. The works of [18, 32] investigated the effects of several optimizations to a multithreaded PDES simulator on smaller-scale platforms such as Intel's Core-i7 and AMD's Magny-cours.

Another area of research involves removing boundaries on resource allocation in a "share-everything" system [15]. Such a system may allow a synchronous system to compete with optimistic methods in unbalanced situations by shifting hardware resources to more highly-loaded LPs. In addition, lock-free or wait-free event queues [13], may improve performance in situations where remote percentages are high.

There has also been work to control optimism in PDES. Linden [21] utilizes the inter-process communication for disseminating time stamp information of future events. LPs can stop local event processing in case of a possibility of a straggler message in order to reduce the rollback cost. This work exploits the model behavior for controlled optimism. On the other hand, we leverage the PDES engine and GVT computation to accelerate the optimistic processing.

The work of [3] is the follow-up to [4], reporting impressive event processing rates on Sequoia BlueGene/Q supercomputer. The recent effort of [7] evaluated PDES performance on Knights Corner processor. The main conclusion of [7] is that Knights Corner does not outperform the host Xeon processor in terms of event rate unless vector units are fully utilized, and increasing the number of threads does not alter that trend. The reasons behind such subpar performance are slower in-order cores and limited amount of physical memory on the accelerator card.

Several other studies investigated the performance of various parallel applications on Xeon Phi (Knights Corner) platforms [14, 22, 24, 26, 27, 34]. However, all of these applications are very different from PDES and in general offer more parallelization opportunities. Evaluating PDES on KNL provides an insight of how similar fine-grain communication-dominated applications will be expected to perform on these platforms.

8 CONCLUDING REMARKS

In this paper, we analyzed the implications of synchronous and asynchronous GVT algorithms on PDES performance on a cluster of Knights Landing processors. We made three main contributions. First, we showed that providing a dedicated MPI communication thread and relieving this thread from normal event processing significantly improves performance compared to the situation when the MPI thread is also tasked with event processing duties. Second, within this dedicated MPI thread model, we showed that a synchronous GVT algorithm based on barrier synchronization typically outperforms asynchronous algorithms for communication-dominated scenarios with frequent rollbacks. At the same time, we demonstrate that asynchronous GVT algorithm provides a significantly better performance for computation-dominated scenarios with high execution efficiency and relatively low rollback rate. Third, we propose a adaptive GVT algorithm (called CA-GVT) that adjusts to the better performing GVT under a given operating conditions. The idea is to augment asynchronous algorithm with additional synchronization when the simulation efficiency is low or the occupancy of the MPI queue is high. We demonstrated that CA-GVT outperforms other GVT algorithms at mixed communication-computation models.

9 ACKNOWLEDGMENTS

This material is based upon work supported by the AFOSR under Award No. FA9550-15-1-0384 and DURIP award FA9550-15-1-0376.

REFERENCES

[1] A. Sodani amd R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. HUtsell, R. Agarwal, and Y. Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. In *IEEE Micro*.

[2] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. 2015. MPI+ threads: Runtime contention and remedies. *ACM SIGPLAN Notices* 50, 8 (2015), 239–248.

[3] Peter D Barnes Jr, Christopher D Carothers, David R Jefferson, and Justin M LaPre. 2013. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 327–336.

[4] D. Bauer, C. Carothers, and A. Holder. 2009. Scalable Time Warp on Bluegene Supercomputer. In *Proc. of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*.

[5] C. Carothers, D. Bauer, and S. Pearce. 2000. ROSS: A High-Performance, Low Memory, Modular Time Warp System. In *Proc of the 11th Workshop on Parallel and Distributed Simulation (PADS)*.

[6] K. M. Chandy and L. Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems* 3, 1 (Feb. 1985), 63–75.

[7] H. Chen, Y.Yao, and W. Tang. 2015. Can MIC Find Its Place in the World of PDES?. In *Proceedings of International Symposium on Distributed Simulation and Real Time Systems (DS-RT)*.

[8] Gabriele D'Angelo, Stefano Ferretti, and Moreno Marzolla. 2012. Time Warp on the Go. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS '12)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 242–248. http://dl.acm.org/citation.cfm?id=2263019.2263057

[9] Ali Eker, Barry Williams, Nitesh Mishra, Dushyant Thakur, Kenneth Chiu, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2018. Performance Implications of Global Virtual Time Algorithms on a Knights Landing Processor. In *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 1–10.

[10] R. Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53.

[11] R. Fujimoto. 1990. Performance of Time Warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulation* 22, 1 (Jan. 1990), 23–28.

[12] G.Chrysos. 2012. Intel Xeon Phi x100 Family Coprocessor - the Architecture. In *Intel white paper*.

[13] S. Gupta and P. A. Wilsey. 2014. Lock-Free Pending Event Set Management in Time Warp. In *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*.

[14] A. Heinecke, K. Vaidanathan, M. Smelianskiy, A. Kobutov, R. Dubtsov, G. Henri, A. Shet, G. Chrysos, and P. Dubey. 2013. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems based on Intel Xeon Phi Coprocessor. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*.

[15] M. Ianni, R. Marotta, D. Cingolani, A. Pellegrini, and F. Quaglia. 2018. The Ultimate Share-Everything PDES System. In *2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 73–84.

[16] M. Ianni, R. Marotta, A. Pellegrini, and F. Quaglia. 2017. A non-blocking global virtual time algorithm with logarithmic number of memory operations. In *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 1–8. https://doi.org/10.1109/DISTRA.2017.8167662

[17] Deepak Jagtap, Ketan Bahulkar, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2012. Characterizing and Understanding PDES Behavior on Tilera Architecture. In *Workshop on Principles of Advanced and Distributed Simulation (PADS 12)*.

[18] D. Jagtap, N.Abu-Ghazaleh, and D.Ponomarev. 2012. Optimization of Parallel Discrete Event Simulator for Multi-core Systems. In *International Parallel and Distributed Processing Symposium*.

[19] D. Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 405–425.

[20] Z. Lin and Y. Yao. 2015. An asynchronous GVT computing algorithm in neuron time warp-multi thread. In *2015 Winter Simulation Conference (WSC)*. 1115–1126. https://doi.org/10.1109/WSC.2015.7408238

[21] Jonatan Linden, Pavol Bauer, Stefan Englblom, and Bengt Jonsson. 2019. Exposing Inter-process Information for Efficient PDES of Spatial Stochastic Systems on Multicores. *ACM Transactions on Modeling and Computer Simulation* 29, 2, 0–25.

[22] M. Lu, L. Zhang, H. Hyunh, Z. Ong, Y. Liang, B. He, R. Goh, and R. Huynh. 2013. Optimizing the MapReduce Framework on Intel Xeon Phi Coprocessor. In *Proceedings of International Conference on Big Data*.

[23] F. Mattern. 1993. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *J. Parallel and Distrib. Comput.* 18, 4 (Aug. 1993), 423–434.

[24] G. Misra, N. Kurkure, A. Das, M.Valmiki, S. Das, and A. Gupta. 2013. Evaluation of Rodinia Codes on Intel Xeon Phi. In *Proceedings of the 4th International Conference on Intelligent Systems, Modelling and Simulation*.

[25] Alessandro Pellegrini and Francesco Quaglia. 2014. Wait-free global virtual time computation in shared memory timewarp systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*. IEEE, 9–16.

[26] S. Pennycook, C. Hughes, M. Smelianskiy, and S. Jarvis. 2013. Exploring SIMD for Molecular Dynamics Using Intel Xeon Processor and Intel Xeon Phi Coprocessors. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*.

[27] A. Ramachandran, J. Vienne, R. Wijmgaart, L. Koesterke, and I. Sharapov. 2013. Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In *Proceedings of International Conference on Parallel Processing (ICPP)*.

[28] B. Samadi. 1985. *Distributed Simulation, Algorithms and Performance Analysis*. Ph.D. Dissertation. Computer Science Department, University of California, Los Angeles, CA.

[29] Jeff S. Steinman. 1993. Breathing Time Warp. In *PADS '93 Proceedings of the seventh workshop on Parallel and distributed simulation*. ACM, 109–118.

[30] Jeff S. Steinman, Craig A. Lee, Linda F. Wilson, and David M. Nicol. 1995. Global virtual time and distributed synchronization. In *Proceedings 9th Workshop on Parallel and Distributed Simulation (ACM/IEEE)*. IEEE, 139–148.

[31] Jingjing Wang, Ketan Bahulkar, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2013. Can pdes scale in environments with heterogeneous delays?. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 35–46.

[32] Jingjing Wang, Deepak Jagtap, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2014. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1574–1584.

[33] Barry Williams, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Philip Wilsey. 2017. Performance characterization of parallel discrete event simulation on knights landing processor. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 121–132.

[34] Biwei Xie, Xu Liu, Jianfeng Zhan, Zhen Jia, Yuqing Zhu, Lei Wang, and Lixin Zhang. 2015. Characterizing Data Analytics Workloads on Intel Xeon Phi. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 114–115.