Demand-Driven PDES: Exploiting Locality in Simulation Models

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

Traditional parallel discrete event simulation (PDES) systems treat each simulation thread in the same manner, regardless of whether a thread has events to process in its input queue or not. At the same time, many real-life simulation models exhibit significant execution locality, where only part of the model (and thus a subset of threads) are actively sending or receiving messages in a given time period. These inactive threads still continuously check their queues and participate in simulation-wide time synchronization mechanisms, such as computing Global Virtual Time (GVT). This wastes resources, ties up CPU cores with threads that offer no contribution to event processing and limits the performance and scalability of the simulation.

In this paper, we propose a new paradigm for managing PDES threads that we call Demand-Driven PDES (DD-PDES). The key idea behind DD-PDES is to identify threads that have no events to process and de-schedule them from the CPU until they receive a message requiring event processing. Furthermore, these inactive threads are also excluded from participation in the GVT computation, accelerating that process as a result. DD-PDES ensures that the CPU cycles are mostly spent on actual event processing, resulting in performance improvements. This architecture allows for significant over-subscription of threads by exceeding the number of available hardware thread contexts on the chip. We demonstrate that on a Knights Landing processor, DD-PDES significantly outperforms the traditional simulation equipped with the best currently proposed GVT algorithms.

CCS CONCEPTS

• **Computing methodologies → Massively parallel and high-performance simulations**; **Discrete-event simulation**; • **Computer systems organization → Multicore architectures**.

KEYWORDS

Parallel Discrete Event Simulation; Global Virtual Time; Intel Xeon Phi; Knights Landing; Manycore Architecture; Performance; Simulation; Modeling; Locality

ACM Reference Format:
Ali Eker, Barry Williams, Kenneth Chiu, and Dmitry Ponomarev. 2020. Demand-Driven PDES: Exploiting Locality in Simulation Models. In *2020 SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS'20), June 15–17, 2020, Miami, FL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3384441.3395976

1 INTRODUCTION

Traditional Parallel Discrete Event Simulation (PDES) engines are designed based on the principle that all simulation threads (in a thread-based implementation) or processes (in a process-based implementation) are always actively involved in the simulation loop and perform basic simulation functions regardless of whether they have events to process or send out, or not. In particular, upon entering the simulation loop, each thread (or process) checks its input queue to process currently scheduled events, updates the simulation state, schedules out events destined for other threads, and performs necessary steps to compute Global Virtual time (GVT) to ensure proper system-wide synchronization and fossil collection. This model simplifies the simulation engine development process, as all threads are treated uniformly regardless of their dynamic operating conditions.

Unfortunately, this "always-on" philosophy to PDES engine design results in significant performance inefficiencies for simulation models that exhibit substantial *temporal execution locality* where only a small subset of simulation threads are active at a given time period. Consider, for example, a war-gaming simulation of a large battlefield operation. If the military actions and corresponding events occur on a localized segment of the battlefield in a given time period, then only the threads responsible for modelling activities in that region would be active. The rest of the threads (for example, the ones modelling tanks or airplanes that are not participating in this particular operation) will be inactive and will have no events to send or receive. As another example, consider parallel VHDL simulation of a modern CPU. Circuitry to handle reset and interrupt processing are exercised infrequently compared to instruction fetch and decode. Even elements such as the floating-point pipeline and last-level caches are not active for a sustained period of time. Most large-scale simulation scenarios will exhibit similar behavior, where only a part of the model is active at a given time.

If a model exhibits high temporal locality, the "always-on" approach to simulation results in a clear waste of execution cycles for several reasons. First, all threads constantly check their input queues, even if these queues are mostly empty. This wastes the CPU cycles without contributing to the committed event rate and simulation throughput. Second, without introducing new events or messages, all these inactive threads still participate in the GVT computation to ensure global synchronization, thus significantly slowing the GVT update process, as its performance depends on the number of participating threads.

To address these inefficiencies, in this paper we propose an alternative approach to PDES engine design which we call *Demand-Driven PDES (DD-PDES)*. The main idea behind DD-PDES is to identify threads that have no events to process or messages to send and de-schedule them from the CPU until they receive a message requiring event processing. Furthermore, these inactive threads are also excluded from participation in GVT computation, accelerating that process as a result. Upon message reception, inactive threads are re-integrated back in the simulation. As a result, DD-PDES ensures that the CPU cycles are mostly spent on actual event processing, resulting in performance and scalability improvements. Furthermore, this architecture allows for significant over-subscription of threads by exceeding the number of available hardware thread contexts on a many-core chip. This, in turn, allows higher-scale simulations to be performed within the boundary of a single chip, without crossing the network and requiring expensive MPI communication.

To quantify the advantages of DD-PDES, we implemented this framework within multithreaded version of ROSS simulator [5, 20]. We applied the concept of DD-PDES to both a synchronous ROSS engine, where the GVT algorithm is based on barrier synchronization, and also to an asynchronous ROSS engine, where the GVT implementation is based on the recently proposed wait-free GVT algorithm [27]. We demonstrate that on a 64-core Intel Knights Landing processor, DD-PDES significantly outperforms traditional simulation equipped with either a synchronous or an asynchronous GVT algorithm, and the advantages increase as smaller percentage of threads participate in event processing and communication during active phases.

The main contributions and the key results of this paper are:

- We observe that traditional approaches to PDES simulation engines are inefficient for the models that exhibit significant temporal execution locality with only a small subset of threads participating in a simulation at a given time.
- To address this inefficiency, we propose DD-PDES — a mechanism to selectively de-schedule inactive threads from the CPU and also remove them from participation in global synchronization. Upon receiving a message, these inactive threads are reintegrated back into the simulation.
- We evaluate DD-PDES using ROSS simulator on an Intel's 64-core Knights Landing many-core processor and demonstrate that performance improvements in the range of 6% and 65% can be achieved for the models that we evaluated.

The rest of the paper is organized as follows. Section 2 provides a brief PDES background and describes our evaluation methodology and hardware platform. Section 3 describes the details of the DD-PDES architecture. Our results are presented in Section 4, we review the related work in Section 5 and offer our concluding remarks in Section 6.

2 BACKGROUND AND EXPERIMENTAL SETUP

In this section, we briefly review PDES basics necessary to understand the proposed technique, describe ROSS simulator and variants of PHold benchmark used for our studies, and overview the hardware platform (Intel Knights Landing processor) used for our experiments.

2.1 PDES Overview

PDES is a parallel implementation of discrete event simulation, allowing to execute various parts of the simulation model in parallel. When utilizing hardware parallelism in the form of multi-core and many-core processors, PDES offers a promise of a significant speedup compared to sequential simulation. At a high-level, a PDES system can be thought of as a collection of Logical Processors (LPs) that execute concurrently and collectively carry out the simulation model. Each LP maintains its own input queue and local virtual simulation time (LVT). To simulate the interactions between simulated system components, LPs communicate via time-stamped (virtual time) event messages [11, 12, 21]. An event message resembles a state change on the corresponding physical system component. LPs store event messages in their input queues to be processed in the order of time-stamps. Each event processing generates a new event message to be sent to any LP including the sender.

To maintain consistency, LPs have to periodically synchronize. There are two approaches to synchronization - conservative and optimistic. We consider optimistic simulation, where LPs process events from its queue and advance their LVT with the possibility that a straggler event (e.g. event generated in the past from a different LP) may appear in the input queue. This happens because LPs proceed with different speed and their advance of local virtual time is not uniform.

A straggler event violates the causality order and requires the erroneously processed messages to be undone and the simulation state to be rolled-back to a consistent state prior to the straggler. To implement a rollback mechanism, all previous states should be stored because, conceptually a straggler message can arrive at any point. Saving every LP state throughout the simulation involves tremendous memory demand, therefore, a mechanism is needed to determine a lower bound (virtual time) on the earliest state where a straggler message can target, so that memory belonging to the states before this estimated time can be garbage collected. This time is called Global Virtual Time (GVT) which determines a lower bound on all straggler messages at any moment in the system.

2.2 GVT Computation Algorithms

GVT algorithms essentially compute the snapshot of a distributed system so that memory to rollback operations can be freed. Once the memory to rollback an operation is freed, the operation is considered committed. In PDES, GVT is computed periodically to garbage collect unnecessary event histories and perform I/O operations. An ideal GVT value is the minimum of two variables: 1) the minimum time stamp of all in-transit messages (possible stragglers) and 2) the minimum LVT of all LPs in the system.

The GVT interval sets the length of the gap (in virtual time) between two consecutive GVT computations. Frequent GVT computations (small intervals) yield efficient memory usage because of aggressive garbage collection, but incur higher synchronization or computational overheads. There are two fundamental approaches to compute GVT, synchronous and asynchronous.

2.2.1 *Synchronous GVT Algorithm.*

Synchronous GVT algorithms rely on global synchronization points such as pthread_barrier or MPI_barrier routines. At each GVT round, LPs synchronize at the barrier calls and wait for all in-transit messages (possible stragglers) to arrive their destinations. Once there is no in-transit message in the system, an LP computes its LVT by taking the last processed event's time stamp (at this point, straggler messages are incorporated). Finally, the GVT is computed by taking the minimum LVT among the LPs.

Synchronous GVT computation is tightly coupled with the rest of the simulation components such as event processing and messaging. During a GVT round, simulation does not progress since no new events are created. Although synchronous GVT algorithms are relatively simple and easy to implement, they may incur high synchronization overhead, especially when the simulation progress is non-uniform. Faster LPs arrive at global synchronization points earlier and have to wait idle for other slower LPs to catch up. Also, the periodic stopping of the simulation detrimentally impacts the parallel performance. For our experiments, we deployed a widely-used *pthread_barrier* based synchronous algorithm, which we call Barrier GVT.

2.2.2 *Asynchronous GVT Algorithms .*

In contrast, asynchronous GVT algorithms decouple GVT computation from other simulation tasks. GVT is computed asynchronously, in the background, while LPs continue event processing and messaging without global synchronization. Asynchronous GVT algorithms estimate a GVT value, therefore they can yield a sub-optimal memory performance. However, they induce minimal synchronization overhead especially in shared memory systems, since asynchronous GVT algorithms are typically implemented as lock-free.

Most asynchronous GVT algorithms rely on constructing consistent cuts along the LPs. A GVT round is composed of multiple phases separated by these consistent cuts. A phase indicates the position of an LP and its messages with respect to the cut. A consistent cut is used to capture the time-stamps of the in-transit messages sent during the preceding phase. The recorded in-transit messages is a subset of all in-transit messages in the system. Based on this subset, a GVT estimation is performed. More advanced asynchronous GVT algorithms utilize multiple cuts to count the number of in-transit messages or to determine the optimal time to trigger the estimation. However, the objective is the same: to produce an estimation as close to the perfect GVT as possible, while inducing minimal synchronization and computational overheads. For our experiments, we implemented the Wait-Free GVT Algorithm, the best performing shared memory GVT implementation in the literature [10, 27].

2.3 ROSS Simulator and Benchmarks

Our experiments in this paper are based on ROSS [5] optimistic PDES simulator, specifically on its multithreaded version developed in [20]. In this case, multiple simulation threads are running simultaneously, each implementing one or more LPs. We drive our experiments by a highly configurable PHold benchmark. While PHold is a synthetic benchmark, it allows to implement various configurations of the simulated model, including event processing granularity (EPG), locality, and percentage of remotely generated events. EPG refers to the duration of event processing and can be used to control the balance between computation and communication in our models. The locality factor is used to create localized simulation models, where only a part of the model is active at a given time. We also control the percentage of events that are generated remotely and involve core-to-core communication, in contrast to local events.

In a balanced PHold model, each LP randomly picks a destination LP to send a new message. Time stamp of a new message is calculated by adding a predefined "lookahead" value to the Local Virtual Time (LVT) of the sender LP. LVT is determined by the last processed event's time stamp. The total number of events in the system remains constant. We modified the original PHold model to create a workload imbalance among the simulation threads. Specifically, some threads are chosen as message destinations more frequently while others rarely receive a new message. The group of active threads shifts as simulation progresses. For our experiments, we implemented models where only half of the threads, one quarter of the threads, one tenth of the threads and one twentieth of the threads are actively communicating with each other in a given time period. We refer to these models 1-2, 1-4, 1-10 and 1-20 respectively. For example, in a 1-2 model, during the first half of the simulation, only the first half of threads communicate within each other. In the second half, communication shifts to the second group. These models serve as a representation for the real-life models that exhibit communication locality.

2.4 Experimental Setup and Metrics

We run our experiments on Intel's second-generation Xeon Phi processor, the Knights Landing (KNL) [13, 31]. KNL can be used as a standalone processor and it contains up to 72 cores, each supporting 4-way hyper-threading. Each KNL core implements branch prediction, out-of-order execution logic and contains 2 Vector Processing Units. Cores are paired into tiles, each with a 1 MB L2 cache. KNL systems also include a 16 GB on-package fast RAM, called MCDRAM. This is a high bandwidth memory (HBM) which can be used as an L3 cache to the off-package DDR4 2400 main memory. We conducted our experiments on a KNL Model 7230 with 96 GB of memory and 64 cores. Our system runs CentOS 7.2 and utilizes GNU Compiler Toolchains by enabling all the optimizations.

We assigned 128 LPs to a simulation thread. LPs are partitioned in a round-robin fashion and each LP simulates a node in the PHold model. LPs are initialized with 1 starting event and they can process and send a batch of 8 events at once. We used *weak scaling* [3] when reporting our performance results. As we increase the thread count, we maintain the number of starting events per thread, thus proportionately increasing the number of total events. We also used the committed event rate to evaluate the performance of a system. Committed event rate is the number of events committed over the total number of events executed per second. The number of committed events are calculated by subtracting the number of rollbacked events from the total number of events.

Finally, we leverage the PAPI profiling tool [32] to observe the hardware performance counters on our KNL chip for a selected portion of the code only. We examine the core simulation loop of ROSS since it maintains the event processing, communication, GVT and DD-PDES components. In addition, we utilize rdtsc (Read Time-Stamp Counter) x86 assembly instructions to determine the CPU time spent on a specific function. This is used to analyze the synchronization overheads of GVT functions.

3 DEMAND-DRIVEN PDES

DD-PDES is motivated by the observation that simulation threads that do not actively participate in message sending and receiving during a period of execution waste CPU resources by checking on the status of their input queues in vain. Since no new messages are discovered, these checks do not contribute to the increases in the event processing rates. In addition, these threads also participate in the GVT algorithm, making the GVT computation process more expensive as more threads are participating in a barrier synchronization or competing for locks, depending on the implementation. One can envision a design where a proper GVT can be computed without contributions from these threads.

We propose a novel thread to core scheduling algorithm which forms the basis of what we call Demand-Driven PDES (DD-PDES). The main objective of DD-PDES is to sustain maximum resource utilization and minimum synchronization overhead throughout the simulation. DD-PDES identifies the threads which do not contribute to the simulation progress but waste hardware resources and incur synchronization overhead. The identified "inactive" threads are scheduled out or "deactivated" to free the resources and to reduce the synchronization overhead. When these inactive threads receive messages, they are woken up and reintegrated back into the simulation. In the following subsections, we describe DD-PDES architecture, examine its components with implementation details and finally characterize performance benefits and overheads.

3.1 CPU Utilization of Inactive Threads

First, we note that some de-scheduling of inactive threads will naturally happen in a PDES system built around Barrier GVT synchronization that uses the pthread_barrier calls in the Barrier GVT computation phase. This is implemented in ROSS-MT simulator used for this study. In an imbalanced system, threads which rarely receive a message start the GVT computation earlier compared to the threads which are busy with event processing because of frequent message receptions. During a GVT computation, these faster inactive threads are yielding the CPU and are scheduled out at pthread_barrier calls until other active threads also start GVT computation and synchronize at pthread_barrier calls. In summary, the inactive threads already get scheduled out at each GVT round as a side effect of the Barrier GVT algorithm.

Figure 1 shows the CPU utilization of ROSS-MT with Barrier GVT synchronization. While almost 100% CPU utilization is recorded for balanced models where all threads are equally occupied with event event processing, the utilization significantly decreases, as the imbalance in computational load and communication frequencies between different threads widens. For example, for the imbalanced model denoted as 1-20 (one in twenty threads is active, we observe only average 8% CPU utilization.

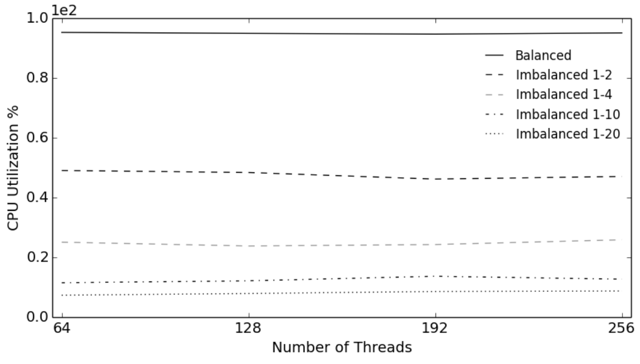

Figure 1: CPU Utilization For Balanced and Imbalanced Models in Baseline ROSS Simulator with Barrier-based GVT

While de-scheduling the threads waiting at barrier calls helps in increasing CPU utilization (as long as there is other work to be performed or other threads to be scheduled), all inactive PDES threads de-scheduled at the barrier will still need to be brought back after the GVT cycle completes and will continue checking their queues until the next GVT round. So, in the big picture, de-scheduling only at GVT intervals has almost no positive impact on the actual performance. As we demonstrate later, when we use an over-subscribed model with significantly more simulation threads than hardware thread contexts on the chip, the simulation throughput immediately degrades, as all threads are still competing for limited CPU core resources frequently.

In contrast, the DD-PDES framework that we propose de-schedules inactive threads systematically and keeps them inactive for a long time, until they receive an incoming message. This process is not specifically tied up to GVT cycles. As a result, the oversubscribed models (with larger number of simulation threads compared to the number of hardware thread contexts) can exhibit significant performance improvements with DD-PDES, as we demonstrate in this paper. In the next few subsections, we describe the details of DD-PDES architecture and then we analyze our results.

3.2 DD-PDES Architectural Overview

DD-PDES mechanism is orchestrated by a special "controller" thread. This thread does not participate in regular simulation tasks such as event processing, GVT computation or messaging. The controller's sole responsibility is to activate (schedule in) the "worker" threads when needed. Worker threads are the regular threads that perform event processing. The controller thread constantly checks for appropriate conditions to reactivate a scheduled out thread. On the other hand, worker threads continuously check their input queue sizes to possibly deactivate themselves (de-schedule themselves out of the CPU) if they did not receive a message in a long time period.

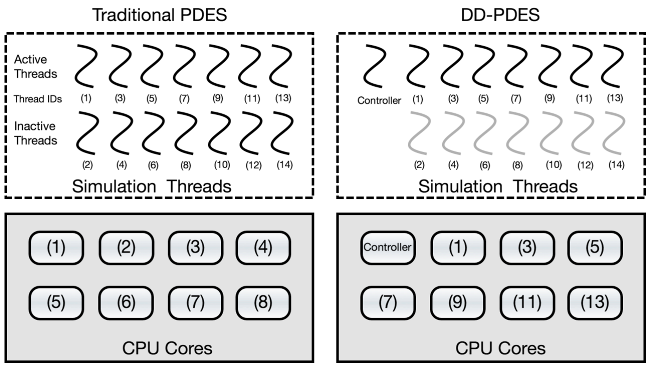

Figure 2: Overview of DD-PDES

We categorize the worker threads as "active" or "inactive" based on the messages in their input queues. An active thread receives frequent messages, performs event processing and contributes to the simulation progress. An inactive thread rarely receives a message, but incurs synchronization overhead and generally wastes hardware resources. Figure 2 outlines (at a high level) the advantages of DD-PDES. A traditional PDES system treats active or inactive threads in the same manner in terms of thread to core scheduling. On the other hand, in a DD-PDES system, the controller schedules out the inactive threads from CPUs while scheduling in the active threads to allow hardware resources to be utilized more effectively. This also accelerates the GVT computation process and reduces the global synchronization overhead since fewer number of threads participate in each. The controller identifies the inactive threads dynamically as simulation evolves. Also in this example, we can see that over-subscription scenarios can be naturally supported by DD-PDES. In this example, fourteen threads (seven active and seven inactive at a given time) can co-exist together and co-execute on only eight physical cores without detrimental effect on the overall event rate. This is not possible with traditional PDES, as we demonstrate in our results section.

Figure 3 presents a simplified DD-PDES system composed of 4 worker threads executing at wall clock time. In the beginning, the simulation is balanced and worker threads uniformly communicate as shown by the dotted arrows. However, because of a state change in the simulated model, communication load shifts towards threads 1 and 2. When this shift is detected by the DD-PDES mechanism, threads 3 and 4 deactivate as they no longer receive messages, depicted as white circles. Later, the communication load shifts in the opposite direction. As threads 3 and 4 receive a message, they reactivate and integrate back into the simulation which is shown as black circles, while threads 1 and 2 deactivate.

Activation and deactivation of worker threads are implemented using binary semaphores. Each worker thread shares a semaphore

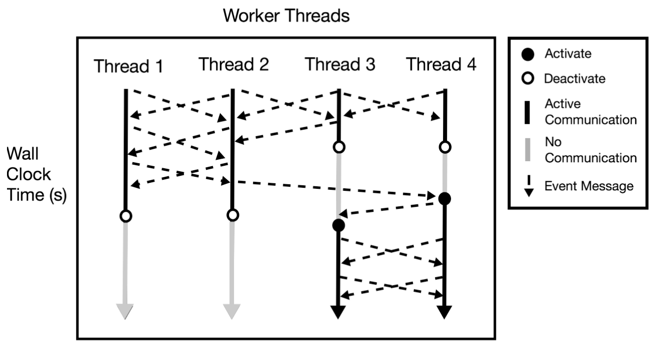

Figure 3: DD-PDES Timing Diagram

with the controller thread. Controller initializes all the semaphores and locks them before worker threads start the simulation. The controller thread also runs on a dedicated CPU core which is not available for scheduling worker threads.

3.3 Worker Thread Deactivation

At each iteration of the core simulation loop, worker threads execute the "read_message_count()" procedure. In this routine, they check their input queue sizes. If the size is read as 0 (meaning they did not receive a new message) a predetermined number of times in a row (as established by the "zero_counter_threshold"), the "blocked" flag for the corresponding thread is turned on, as shown in Algorithm 1, lines 1 to 8.

Algorithm 1 Self-Deactivation of Worker Threads

```
1:  procedure READ_MESSAGE_COUNT
2:      if !$worker.blocked$ then
3:          if $input\_queue[W\_id].size = 0$ then
4:              $worker.zero\_counter$ += 1
5:          else
6:              $worker.zero\_counter \leftarrow 0$
7:          if $worker.zero\_counter > zero\_counter\_threshold$ then
8:              $worker.blocked \leftarrow True$
9:  procedure COMPUTE_GVT
10:     if $GVT\_last\_phase$ and $worker.blocked$ then
11:         $complete\_event\_processing()$
12:         $blocked\_threads[W\_id] \leftarrow True$
13:         $num\_active\_treads$ −= 1
14:         $sem\_wait(sem\_locks[W\_id])$                ▷ $Deactivation$
15:         $num\_active\_threads$ += 1
16:         $blocked\_threads[i] \leftarrow False$
17:         $worker.blocked \leftarrow False$
18:         $worker.zero\_counter \leftarrow 0$
```

A worker thread starts the deactivation process when it checks its "blocked" flag as true at the end of the current GVT round, as shown in Algorithm 1, line 10. Then, the worker thread processes the events in its input queue (if there is any) and sends corresponding messages to the appropriate destinations. This is abstracted out as "complete_event_processing()" function that is shown on line 11.

This process ensures that when a thread is reactivated, it does not contain events belonging to earlier portions of the simulation, since simulation will progress as remaining active threads continue event processing, communication and GVT computations. When the reactivated worker is integrated back into the simulation, it should not pull back the entire system to an obsolete state by processing events that were generated before its deactivation. Messages sent to an inactive worker thread are handled explicitly. This is further explained at the end of subsection 3.5.

The Boolean array of "blocked_threads" is used to indicate which worker threads are active in a given time. During the deactivation process, a worker turns on its flag to inform the controller and decrements the number of active threads (lines 12 and 13). An array of binary semaphores named "sem_locks" is used to actually schedule out and schedule in selected threads. As the worker calls "sem_wait" on its semaphore (line 14), it deactivates until the controller calls "sem_post" using the same semaphore. When the "sem_post" is issued, the reactivated worker resets the necessary variables (lines 16 to 18) and integrates back into the simulation.

3.4 Worker Thread Activation

The controller thread does not participate in regular PDES tasks, but instead continuously loops to possibly reactivate worker threads. The controller activates a worker thread based on two scenarios. First, if the number of active threads is zero (which can be the case with a small "zero_counter_threshold" such as 10) the controller activates each worker thread one by one as shown in Algorithm 2, lines 3 to 5.

Algorithm 2 Activation of Worker Threads By the Controller

```
 1: procedure CONTROLLER_EXECUTE
 2:     while !$simulation\_done$ do
 3:         if $num\_active\_threads = 0$ then
 4:             for $i \leftarrow 1$ to $num\_workers$ do
 5:                 $sem\_post(sem\_locks[i])$                        ▷ Activation
 6:         if $num\_active\_threads < num\_worker\_threads$ then
 7:             if $poll\_counter = 0$ then
 8:                 if !$gvt\_in\_progress()$ then
 9:                     $enter\_critical()$
10:                     for $i \leftarrow 1$ to $num\_workers$ do
11:                         if $blocked\_threads[i]$ then
12:                             if $input\_queue[i].size > inq\_threshold$ then
13:                                 $sem\_post(sem\_locks[i])$        ▷ Activation
14:                     $critical\_done()$
15:                     $poll\_counter \leftarrow poll\_threshold$
16:             else if $poll\_counter > 0$ then
17:                 $poll\_counter$ -= 1
```

A thread can also be reactivated under the following three conditions: 1) the number of active worker threads is less than the total number of worker threads (line 6); 2) the "poll_counter" exceeds the "poll_threshold" (lines 7 and 15 to 17); and 3) GVT computation is not currently in progress (line 8). Worker threads should not be activated during a GVT computation to ensure the integrity between GVT variables and phases. Specifically, the in-transit message counters (Barrier GVT) and the phase counters (Wait-Free GVT) are updated at the end of a GVT round based on the new number of active threads. Therefore, worker threads can only be activated or deactivated at the end of a GVT round.

If all three conditions above are satisfied, the controller enters a critical section protected by "compare_and_swap" atomic operation utilized in enter_critical() routine (line 9). In this critical section of code, the controller scans the "blocked_threads" array and checks each inactive worker thread's input queue size. If the size exceeds "inq_threshold", the blocked thread is reactivated by issuing "sem_post" on its semaphore as shown in lines between 10 and 13.

In our experiments, we set the "polling_threshold" to 1000 and "inq_threshold" to 0. While the former one sets up an activation frequency, the latter one guarantees the activation upon a single message reception. For the "zero_counter_threshold" which determines how aggressive the workers deactivate, a range of values between 100 and 10000 is tested and the best performing value for each thread count and simulation model is chosen. Similarly, to set the GVT interval, we tested a range of intervals between 50 and 3200 to determine the best performing interval per thread count and simulation model for each algorithm.

3.5 Synergy of DD-PDES with GVT Algorithms

We implemented two DD-PDES models: *DD-PDES-Sync*, which runs on top of a synchronous GVT algorithm (Barrier GVT) and *DD-PDES-Async* which works with the best performing asynchronous GVT algorithm (Wait-Free GVT). Both GVT implementations had to be modified to work seamlessly with the corresponding DD-PDES model. For *DD-PDES-Sync*, we implemented custom, spin-wait barrier calls with a variable number of participating threads using atomic operations. This ensures that GVT computation does not schedule out the threads waiting on the barrier calls, but instead, it is only the DD-PDES mechanism that deactivates and activates the worker threads. For *DD-PDES-Async*, an extra switch statement is added to the last phase of Wait-Free GVT algorithm (Phase End). Using this switch, a thread deactivates itself if it checks its "blocked" flag as true. Also, counters are added to manage the number of active threads participating in GVT phases. Similar counters and a switch statement also present in the *DD-PDES-Sync*.

In order to compute a correct GVT value, messages sent to inactive threads must be taken care of explicitly if "inq_threshold" is larger than 0. Specifically, worker threads record the time stamps of the messages they send to inactive threads. During the GVT computation, a worker thread computes the minimum time stamp among these messages. This time stamp is incorporated into the GVT computation to ensure the monotonic progress of the GVT function. For example, when a worker thread W is inactive, the simulation advances and a new GVT value of T is computed. When W is reactivated, it has to process all messages destined to it during its inactive period. Based on these messages, W can compute its LVT as smaller than T. This violates the monotonic GVT progress. W can also generate straggler messages with time stamps prior to T which breaks the GVT premise. Theoretically, a message sent to an inactive thread will keep the GVT constant if the inactive thread is never activated again. This never happens as we set the "inq_threshold" to 0.

3.6 Performance Benefits & Overheads

When a worker thread W is deactivated, it is scheduled out of the CPU. This frees up the hardware resources belonging to W to be utilized by threads which contribute to the simulation progress more effectively. Deactivation also reduces the global synchronization overhead since fewer threads compete for shared resources. GVT computation also involves less threads, further accelerating execution. These are the main performance benefits of the DD-PDES mechanism.

The DD-PDES paradigm also comes with its overheads. At least one worker thread is lost to reserve a CPU core for the controller. Shared data structures assisting DD-PDES can cause communication overhead between threads residing on separate cores. However, this can be compensated by reducing the global synchronization and by better utilizing hardware resources.

The DD-PDES mechanism for sharing data structures is lock-free. It utilizes hardware atomic operations to protect the critical sections between the controller and the worker threads. Specifically, atomic *add* and *sub* operations are used to modify the shared counters and a *compare_and_swap* operation is used to protect the semaphores and the shared arrays.

4 RESULTS AND DISCUSSION

Recent research demonstrated that synchronous GVT algorithms based on barrier synchronization performs better than asynchronous GVT algorithms for imbalanced models (as extra synchronization puts a cap on the degree of disparity in threads' progress), while asynchronous GVT algorithms performs better for balanced models [10]. Therefore, we demonstrate two DD-PDES models: *DD-PDES-Sync* and *DD-PDES-Async* which are implemented on top of a synchronous Barrier GVT and an asynchronous Wait-Free GVT algorithm [27], respectively. We evaluate our DD-PDES implementations against the corresponding baseline models: *Baseline-Sync* and *Baseline-Async*, which support Barrier GVT and Wait-Free GVT respectively, without DD-PDES support. We begin our analysis with a balanced model, and then analyze progressively more imbalanced models.

4.1 Balanced Model

The DD-PDES paradigm is primarily designed for imbalanced models that exhibit significant locality, however it also offers competitive performance in balanced models. As seen in Figure 4, *DD-PDES-Async* is outperformed by *Baseline-Async* by only 1.25% while *DD-PDES-Sync* outperforms *Baseline-Sync* by 10.3%. Note that our KNL model can run 256 hardware threads simultaneously as it contains 64 cores, each supporting 4-way hyper-threading.

Since no activations or deactivations actually occur in these models and all threads are always active, a small performance degradation of *DD-PDES-Async* compared to *Baseline-Async* indicates the overhead of checks that are performed for activation and deactivation conditions. This results show that this overhead is minimal and our current implementation is quite light-weight. For example, in a 256-way simulation, the average CPU time for a GVT round is 3.47 and 3.48 seconds for *Baseline-Async* and *DD-PDES-Async*, respectively. The most significant disadvantage of DD-PDES mechanism is that it is sacrificing one of the CPU cores to run the controller thread. In traditional designs, all cores can be occupied by the worker threads. The reason that *DD-PDES-Sync* outperforms *Baseline-Sync* stems from its lighter, customised spin-wait barrier calls.

4.2 Moderately Imbalanced Models

As seen in Figure 5, in the 1-2 Imbalanced Model (half of the threads are active at one time) and 256-way simulation, *DD-PDES-Async* outperforms *Baseline-Async* and *Baseline-Sync* models by 31.6% and

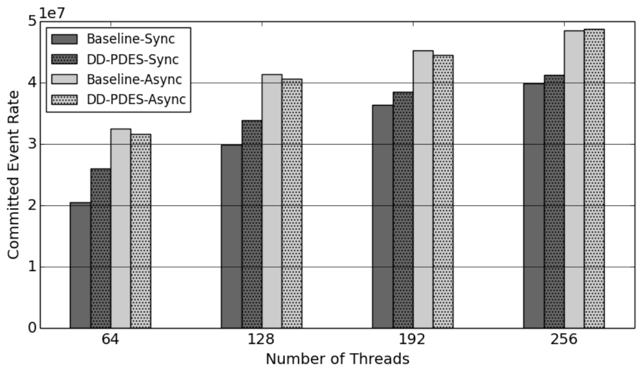

Figure 4: Balanced Model

5.6%, respectively. The performance gap increases to 64.7% and 19.6% in the 1-4 Imbalanced Model (quarter of all threads are active in a given time period) as shown in Figure 6.

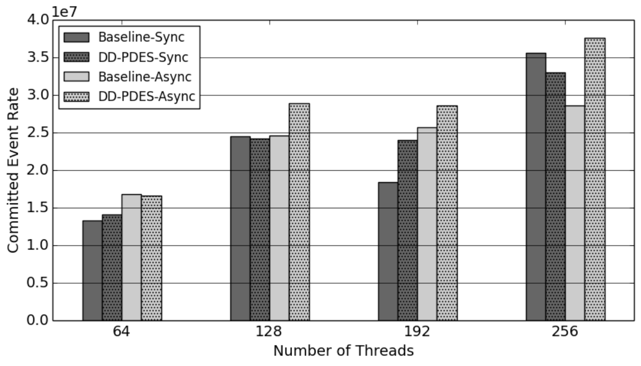

Figure 5: Committed Event Rate for 1-2 Imbalanced Model

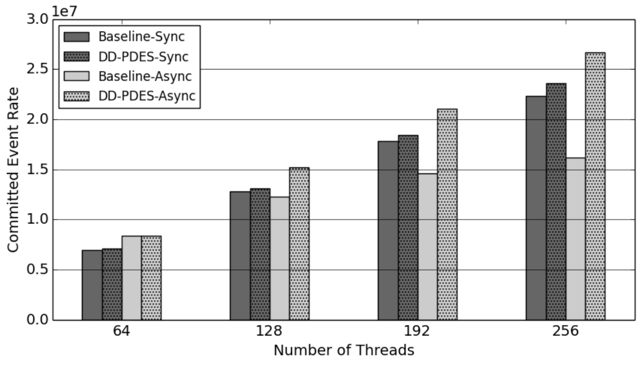

Figure 6: Committed Event Rate for 1-4 Imbalanced Model

Figure 7 presents the timing diagram of DD-PDES execution in the 1-4 Imbalanced Model. As the communicational load shifts between 4 thread groups, a specific group stays active while others deactivate. The activation and deactivation times are depicted as black and white circles, respectively, in terms of which GVT round they happen. As simulation begins with a communicational load on group 1, threads in the other groups deactivate during the GVT round 3. As the communicational load shifts to the next group, the associated threads reactivate and integrate back into the simulation. This process repeats until the last GVT round of 7917 where all inactive threads are reactivated and simulation completes.

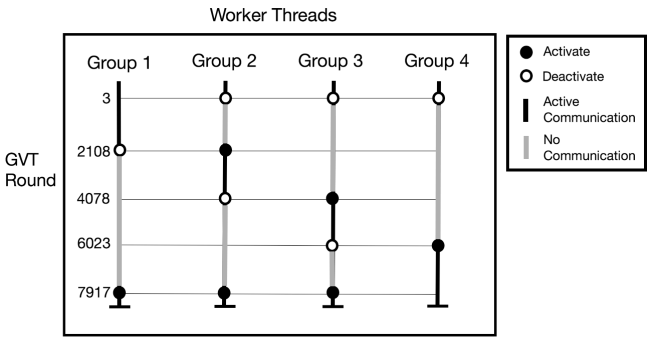

Figure 7: DD-PDES timing diagram for 1-4 Imbalanced Model

We also analyzed the impact that DD-PDES has on the CPU utilization. Using the PAPI tool, we compute the total number of instructions issued and cycles executed during the core simulation loop of ROSS. Then, we calculate the average CPI among cores to evaluate the overall utilization by dividing total cycles by the total instructions.

In the 1-2 Imbalanced Model and 256-way simulation *DD-PDES-Async* executes at the lowest CPI (cycles per instruction) of 1.22, while *Baseline-Async* and *Baseline-Sync* execute at 1.39 and 1.26 CPI, respectively. Similarly, in the 1-4 Imbalanced Model, *DD-PDES-Async* has 0.9 CPI, while *Baseline-Async* and *Baseline-Sync* have 1.4 and 0.96 CPI, respectively. *DD-PDES-Async* spends less cycles per instruction by eliminating unnecessary execution which does not contribute to the simulation progress. *Baseline-Sync*'s performance is close to the *DD-PDES-Async* because of the context switches at the pthread_barrier calls during its Barrier GVT computations.

We evaluate the synchronization overhead by computing the average CPU time spent on a GVT computation using rdtsc instructions. DD-PDES models reduce the number of threads participating to the GVT computation thus, accelerating its speed. In the 1-2 Imbalanced Model with 256-way simulation, the average CPU time spent during a GVT computation is 0.39 and 4.31 for *DD-PDES-Async* and *DD-PDES-Sync*, respectively. However, these number are 6.74 and 18.35 seconds for *Baseline-Async* and *Baseline-Sync*. As almost double the number of GVT computations are performed in the 1-4 Imbalanced Model compared to the 1-2 Model (7917 - 4010), each GVT round takes less time. Nevertheless, an average GVT round lasts 0.27 and 1.67 seconds for and *DD-PDES-Sync*, respectively while for *Baseline-Async* and *Baseline-Sync*, these numbers are 0.78 and 10.86 seconds.

4.3 Highly Imbalanced Models

The Highly Imbalanced Models contain a smaller number of active threads at a given time compared to Moderately Imbalanced Models. For example, in 1-10 and 1-20 Models, only 10% and 5% of the threads actively contribute to the simulation progress, respectively. As stated earlier, synchronous GVT algorithms are more advantageous at imbalanced models due to the smaller disparity between simulation threads caused by global periodic synchronizations [10]. Therefore, as seen in Figures 8 and 9, *DD-PDES-Sync* performance gains are significant as it outperforms *Baseline-Sync* and *Baseline-Async* by 88% and 1077% in the 1-10 Imbalanced model, and by 63% and 785% in the 1-20 Imbalanced model, respectively. These graphs also demonstrate scalability up to 1024 threads.

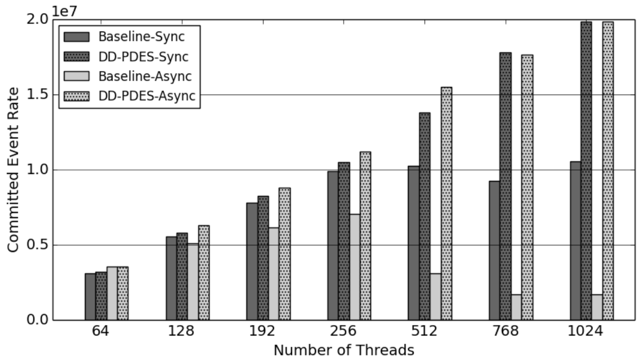

Figure 8: Committed Event Rate for 1-10 Imbalanced Model

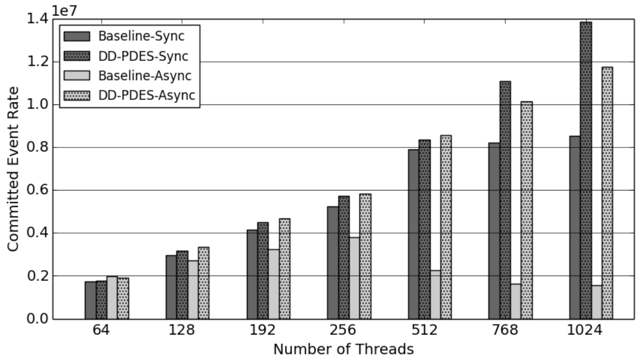

Figure 9: Committed Event Rate for 1-20 Imbalanced Model

DD-PDES-Sync can execute the same number of instructions in about one half and one third of the cycles required by *Baseline-Sync* and *Baseline-Async*, respectively. Specifically, *DD-PDES-Sync* executes at 0.27 and 0.24 CPI while *Baseline-Sync* executes at 0.55 and 0.44 CPI in 1-10 and 1-20 Imbalanced Models, respectively. These numbers are 0.77 and 0.7 for *Baseline-Async*.

DD-PDES significantly accelerates the GVT computation by eliminating the unnecessary involvement of a large number of threads. For example, at a 1024-way simulation and in 1-10 Imbalanced

Model, an average CPU time spent during a GVT round for *DD-PDES-Sync* is 8.00 seconds while it is 43.86 and 324.43 seconds for *Baseline-Sync* and *Baseline-Async*, respectively. Similarly, in 1-20 Imbalanced Model, *DD-PDES-Sync* completes a GVT round at 5.69 seconds on average while *Baseline-Sync* and *Baseline-Async* do so at 94.15 and 217.96 seconds.

4.3.1 Over-subscription Analysis. *Baseline-Async* demonstrates significant performance degradation in the over-subscription scenarios as more threads are supported in the simulation than the number of hardware thread contexts, and threads effectively multiplex for the use of limited contexts by exploiting temporal execution locality and relatively long inactivity periods of some threads. In addition, *DD-PDES-Sync* outperforms *DD-PDES-Async* because of its barrier based GVT implementation which decreases the disparity and increases the efficiency in Highly Imbalanced Models. For example, in a 1024-way simulation, *DD-PDES-Sync* achieves 94.58% and 93.05% efficiency while *DD-PDES-Async* has 74.9% and 73.63% efficiency in 1-10 and 1-20 Imbalanced Models, respectively.

4.4 Results Summary

When the simulation is moderately imbalanced (models 1-2 and 1-4), *DD-PDES-Async* outperforms the Baseline implementations up to 65% as balanced models are more suitable for asynchronous computation. However, when the imbalance is wider as in the case of models 1-10 and 1-20, *DD-PDES-Sync* performs significantly better as synchronous GVT is more effective for an imbalanced system, where some threads have significantly higher event processing and communication load. Nevertheless, both DD-PDES models improve the performance of *Baseline-Sync* and *Baseline-Async* at both imbalanced and balanced models.

Furthermore, the DD-PDES paradigm creates the over-subscription opportunities, where more threads can execute on the same many-core chip than the number of hardware thread contexts that are available. The hardware resources are virtually over saturated by spawning more simulation threads than a processor can simultaneously support. Our KNL processor contains 256 hardware threads but in some cases, we demonstrate scalability for up to 1024 threads by exploiting temporal execution locality in the simulation model. In such cases, *DD-PDES-Sync* achieves up to 11X performance improvements against the baseline implementations.

5 RELATED WORK

Numerous PDES optimizations have been proposed in the literature for many-core architectures. The main objective is to leverage parallelism obtained by a large number of threads on a shared memory system. Jagtap et al. [19] examined PDES performance on Tilera architecture. Authors in [7, 10, 34] investigated PDES performance on Intel's Xeon Phi processor while the works of [20, 33] investigated the effects of several optimizations to a multithreaded ROSS simulator on smaller-scale multi-core systems and clusters.

A "share-everything" system proposed in [17] allows a synchronous system to compete with optimistic methods in unbalanced situations by shifting hardware resources to more highly-loaded LPs. Although such a system can effectively allow more than one core to handle the load of a single LP, our proposal involves less drastic changes to the design of core PDES functions. In addition, lock-free or wait-free event queues [14] may improve the performance in situations where large number of threads competing for shared memory resources. A recent work of [4] exploits the Intel x86-64 hardware profiling facilities to accelerate the checkpointing mechanism in a speculative PDES system.

An orthogonal approach to DD-PDES is model partitioning [1] and dynamic reassignment of threads to cores to achieve balanced execution [35]. Even if a perfect balance can be temporarily achieved by partitioning, dynamic changes in the simulation behavior may require dynamic re-balancing of the workload to maintain balanced execution, which is expensive and requires significant changes to the simulator core.

There has also been proposals targeting distributed memory architectures. The work of [2] is the follow-up to [3], reporting impressive event processing rates on Sequoia BlueGene/Q supercomputer. Eker et al. [9] utilizes a cluster of KNL processors and presents the idea of a controlled synchrony. A novel semi-synchronous GVT algorithm is used to manage the optimism level based on the simulation dynamics. Similarly, [23] utilizes the inter-process communication for disseminating time stamp information of future events to control optimism in PDES.

As an efficient GVT computation is indispensable for the PDES scalability, GVT algorithms are studied extensively in the literature. Samadi [30] developed one of the first GVT algorithms and introduced the transient message and simultaneous reporting problems. The same year, Chandy and Lamport [6] describe one of the first distributed snapshot algorithms. Mattern [24] built on that to develop an asynchronous algorithm that does not require acknowledgement messages.

Recent GVT proposals target many-core architectures such as [18, 22]. These projects' objective is to create non-blocking and asynchronous GVT algorithms for shared memory architectures. On the other hand, Perumalla [28, 29] reports scalablity up to hundred of thousands cores by proposing novel GVT algorithms. Researchers in [25, 26] propose adaptive load balancing and dynamic GVT algorithms based on irregular PDES workloads.

Oversubscription of the hardware resources on manycore settings has recently been an important area of research as more applications and libraries utilize highly parallel programming models. Authors in [16] present one of the first evaluations of oversubscription scenarios for applications with various parallel programming models running on multi-socket, manycore processors. This work reports that a performance improvement in the range of 27-46% can be achieved by oversubscribing the CPU cores. A more recent work of [36] oversubscribes cluster nodes by co-locating the applications based on their resource consumption characteristics. Huang et al. [15] presents a new MPI implementation where several virtual MPI processes can be mapped to a single physical core. The work of [8] evaluates the performance implications of resource sharing on SMT processors for OpenMP Programs and conversely reports that oversubscription on SMT cores can lead to performance degradation.

6 CONCLUDING REMARKS

Scaling PDES in a cluster environment is a challenging undertaking, as network communication delays are many orders of magnitude higher than the delays of on-chip communication. Therefore, it is attractive to consider designs that attempt to maximize the scale of simulation that can be performed within a single manycore processor, without crossing the chip boundary. In this paper, we proposed one such design - Demand-Driven PDES (DD-PDES).

The main idea of DD-PDES is to exploit temporal execution locality in simulation models, where only a small subset of all simulation threads are active at a given time. Inactive threads are systematically de-scheduled from the CPU until the time when they receive an event. Upon receiving the event, these threads are reintegrated back to the simulation. We implemented DD-PDES on top of PDES engines that use both synchronous and asynchronous GVT algorithms. Our results show promising performance trends and demonstrate that significant over-subscription is possible when substantial locality in the model exists. DD-PDES can be an instrumental piece in future high-performance PDES framework by allowing to effectively execute larger models on the same many-core chip and reducing the need for off-chip communication.

7 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] K. Bahulkar, J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. 2012. Partitioning on Dynamic Bahavior for Parallel Discrete Event Simulation. In *26th IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulations (PADS)*.

[2] 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.

[3] 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)*.

[4] S. CarnÃă, S. Ferracci, E. De Santis, A. Pellegrini, and F. Quaglia. 2019. Hardware-Assisted Incremental Checkpointing in Speculative Parallel Discrete Event Simulation. In *2019 Winter Simulation Conference (WSC)*. 2759–2770.

[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] Matthew Curtis-Maury, Xiaoning Ding, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2005. An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming* (Eugene, OR, USA) *(IWOMPâĂŹ05/IWOMPâĂŹ06)*. Springer-Verlag, Berlin, Heidelberg, 133âĂŞ144.

[9] Ali Eker, Barry Williams, Kenneth Chiu, and Dmitry Ponomarev. 2019. Controlled Asynchrnous GVT: Accelerating Parallel Discrete Event Simulation on Many-Core Clusters. In *Proceedings of 48th International Conference on Parallel Processing (ICPP 2019)*. 1–10.

[10] 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.

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

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

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

[14] 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)*.

[15] Chao Huang, Orion Lawlor, and L. KalÃľ. 2004. Adaptive MPI. 306–322. https://doi.org/10.1007/978-3-540-24644-2_20

[16] C. Iancu, S. Hofmeyr, F. BlagojeviÄĞ, and Y. Zheng. 2010. Oversubscription on multicore processors. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–11.

[17] 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.

[18] 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

[19] Deepak Jagtap, Ketan Bahulkar, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2012. Characterizing and understanding pdes behavior on tilera architecture. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 53–62.

[20] 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*.

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

[22] 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

[23] Jonatan Linden, Pavol Bauer, Stefan Engblom, 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.

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

[25] Eric Mikida and Laxmikant V Kale. 2018. Adaptive methods for irregular parallel discrete event simulation workloads. In *SIGSIM-PADS 2018 - Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. Association for Computing Machinery, Inc, 189–200. https://doi.org/10.1145/3200921.3200936

[26] Eric Mikida and Laxmikant V Kale. 2019. An adaptive non-blocking GVT algorithm. In *SIGSIM-PADS 2019 - Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 25–36. https://doi.org/10.1145/3316480.3322896

[27] 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.

[28] Kalyan S. Perumalla, Alfred J. Park, and Vinod Tipparaju. 2011. GVT Algorithms and Discrete Event Dynamics on 129K+ Processor Cores. In *Proceedings of the 2011 18th International Conference on High Performance Computing*. IEEE Computer Society, 1–11.

[29] Kalyan S. Perumalla, Alfred J. Park, and Vinod Tipparaju. 2014. Discrete Event Execution with One-Sided and Two-Sided GVT Algorithms on 216,000 Processor Cores. *ACM Trans. Model. Comput. Simul.* 24, 3, Article 16 (June 2014), 25 pages. https://doi.org/10.1145/2611561

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

[31] A. Sodani, 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*.

[32] D Terpstra, H Jagode, H You, and J Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009, 3rd Parallel Tools Workshop*. Springer Berlin / Heidelberg, Dresden, Germany, 157–173.

[33] 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.

[34] 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.

[35] Linda F Wilson and Wei Shen. 1998. Experiments in load migration and dynamic load balancing in speedes. In *1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274)*, Vol. 1. IEEE, 483–490.

[36] Q. Xiong, E. Ates, M. C. Herbordt, and A. K. Coskun. 2018. Tangram: Colocating HPC Applications with Oversubscription. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7.