GVT-Guided Demand-Driven Scheduling in Parallel Discrete Event Simulation

Ali Eker  
aker1@binghamton.edu  
Binghamton University  
Binghamton, NY, USA

David Timmerman  
dtimmer1@binghamton.edu  
Binghamton University  
Binghamton, NY, USA

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

Kenneth Chiu  
kcchiu@binghamton.edu  
Binghamton University  
Binghamton, NY, USA

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

ABSTRACT

The performance and scalability of Parallel Discrete Event Simulation (PDES) can be significantly impacted by temporarily inactive threads that occupy CPU resources but do no useful processing. A recent design called Demand-Driven PDES (DD-PDES) identifies such threads and de-schedules them from CPU cores to eliminate the unnecessary overhead. In this paper, we propose significant further improvements to DD-PDES. First, we introduce a new GVT (Global Virtual Time)-guided algorithm named GG-PDES to perform de-scheduling operations in a lock-free fashion and without relying on a centralized controller thread as was used previously. Second, we introduce the Dynamic CPU Affinity algorithm built on top of GG-PDES that adaptively pins simulation threads to CPU cores to achieve a balanced execution. We demonstrate that these optimizations can yield performance improvements in the range of 13% to 50% over the original DD-PDES system.

CCS CONCEPTS

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

KEYWORDS

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

1 INTRODUCTION

Many real-life models that drive Parallel Discrete Event Simulation (PDES) systems exhibit execution locality, where some simulation threads do not process event messages for a period of time [10]. For example, in epidemiology simulations, hot spots affect only a certain group of agents and later mutate or alternate [28, 30]. In traffic simulations, vehicular activity fluctuates on a daily basis or in emergency situations [14, 21, 35]. These changes can render some components of the simulation temporarily dormant. However, corresponding “idle” threads continue to check their input queues for incoming event messages and participate in global synchronization operations, wasting processor cycles without contributing any useful work towards simulation progress. A recent study proposed a mechanism to exploit this execution locality through the concept of Demand-Driven PDES (DD-PDES) [10]. DD-PDES marks threads as active or inactive based on whether they have event messages to process, and de-schedules inactive threads from CPU cores until they become active again.

Advantages of DD-PDES are three-fold: 1) more effective hardware utilization as the unnecessary execution is eliminated; 2) acceleration of global synchronization operations (GVT computation) because less number of threads compete for synchronization primitives; 3) the possibility to support over-subscription scenarios where a processor is overloaded with more threads than available CPU contexts, assuming that some of these threads remain inactive in a given time.

Although the benefits of DD-PDES are clearly demonstrated, the performance experiments in [10] are all based on synthetic workloads and show limited scalability. We argue that DD-PDES design is sub-optimal as its core operations depend on a dedicated controller thread which creates a bottleneck for large-scale simulations. The controller thread is excluded from core simulation tasks such as event processing. Instead, it exclusively manages DD-PDES operations which need to be serialized by locks.

In this paper, we first propose a new, streamlined DD-PDES design where the centralized implementation built around a dedicated controller thread is replaced with a GVT (Global Virtual Time)-guided algorithm that dynamically selects a coordinator. This optimization enables simulation threads to bear the responsibilities of the controller thread interchangeably in a lock-free fashion, hence mitigating the performance challenges of the original DD-PDES. We call this new design GVT-Guided PDES (GG-PDES).
Second, we propose a Dynamic CPU Affinity algorithm built on top of GG-PDES, that adaptively pins active threads to available CPU cores for balanced workload distribution. We benchmark the Dynamic CPU Affinity approach against the conventional “Constant CPU Affinity” and “No CPU Affinity” algorithms and demonstrate substantial performance improvements.

We show that GG-PDES significantly improves the overall performance of DD-PDES and can scale up to 4096 simulation threads (POSIX) within a single Intel’s Knights Landing processor with 256 hardware threads when sufficient temporal execution locality is present in the simulation model. At the same time, DD-PDES implementation achieves scalability only up to 1024 simulation threads.

Finally, we evaluate GG-PDES on several applications. In addition to the classical PHOLD benchmark (also used in the original DD-PDES work), we use two other PDES applications under the ROSS simulation engine. The first one simulates the spread of epidemics, while the second one simulates vehicular traffic patterns. These two applications provide more realistic workloads.

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

- We propose GG-PDES — a streamlined, GVT-guided DD-PDES approach that eliminates the need for a dedicated controller thread. Instead, simulation threads bear the responsibilities of the scheduling management interchangeably, in a lock-free manner. We show that this new design achieves up to 50% performance improvements over the work of [10] for models where simulation threads do not exceed hardware thread contexts.
- In addition, we demonstrate up to 44% performance increase in over-subscription scenarios where we overload our processor with more simulation threads than available hardware thread contexts.
- We further improve the GG-PDES system by introducing the Dynamic CPU Affinity algorithm that pins active threads to the best available CPU cores to maintain a balanced execution without over-saturating individual cores. The Dynamic CPU Affinity outperforms other affinity schemes by up to 35%.
- In addition to the synthetic PHOLD model, we evaluate GG-PDES on two realistic simulation models: a newly developed (under ROSS simulator) location-aware Epidemics Simulation Model and the Traffic Simulation Model. We demonstrate that performance improvements between 13% and 29% can be realized for these models.

2 BACKGROUND

2.1 PDES Overview

At a high level, PDES can be thought of as a collection of Logical Processes (LPs) that execute in parallel on different threads or processes and conduct simulation tasks collectively. LPs model distinct components of the simulated system and communicate via time-stamped (virtual-time) event messages. An event message denotes a state update on the corresponding system component. Each LP maintains a Local Virtual Time (LVT) to be determined by the last processed event’s time-stamp and a state corresponding to the simulated component [20].

For a consistent PDES system, the causality order between event messages has to be managed explicitly. There are two fundamental approaches to maintain the causality order: the conservative approach, that synchronizes LPs globally and periodically to enforce the correct execution order of event messages at all times; and the optimistic approach, that allows LPs to process their event messages out-of-order (speculatively) [13]. In the optimistic approach, incorrectly executed events are eventually detected and recovered via state-saving and rollback mechanisms.

In optimistic PDES, LPs have to save their state updates to a data structure after the processing of each and every event message. This data structure enables an LP to change its current state to one of its previous states if the LP processed an event message out-of-order. This operation is generally known as the rollback. A rollback occurs when an LP processes a straggler message that has a time-stamp less than the LP’s LVT. A rolled-back LP eventually processes its event messages in the correct order of time-stamps and can also trigger rollbacks on other LPs via anti-messages.

The data structure used to save state updates grows over time and can generate a significant memory demand as LPs continuously process event messages. Thus, LPs periodically and collectively compute the Global Virtual Time (GVT) which determines a lower-bound (virtual-time) for a rollback to target at a given time. States update occurred at a virtual-time prior to the GVT can be garbage collected as it is guaranteed that LPs will not rollback to those states.

There are two approaches to compute the GVT: the synchronous approach, that computes a perfect GVT value and leads to the optimum memory performance by synchronizing LPs using barrier calls at each GVT round; and the more light-weight asynchronous approach, that performs a GVT estimation without synchronizing LPs. Asynchronous algorithms typically divide the simulation into multiple phases based on consistent cuts across LPs to estimate the GVT.

2.2 ROSS Simulator

Rensselaer’s Optimistic Simulation System (ROSS) is a high-performance, highly scalable PDES engine [5]. We use its optimistic (Time Warp) single-process and a multi-threaded version specifically designed for shared memory architectures [19]. Note that previous discussions on PDES background were centered around the concept of LP. In ROSS, a simulation thread (POSIX thread) serves a set of LPs and maintains an input queue to store all of its LPs’ incoming event messages. Threads can process and send a batch of 8 events at each cycle of the core simulation loop. LPs are mapped to threads in a Round-Robin fashion while each LP models a node in the simulation. Our version of ROSS does not employ a static or dynamic load-balancing algorithm. We now describe three different models used in this study: PHOLD, Epidemics, and Traffic Model.

2.3 Simulation Models

2.3.1 PHOLD Model. PHOLD is a traditional and versatile simulation model employed frequently by the PDES community [12]. LPs in PHOLD generate and send one new event based on each event message they receive and process. The time-stamp of the new event is computed by adding a “lookahead” value (drawn from
a probability distribution) to the Local Virtual Time (LVT) of the
sender LP. LPs are each initialized with one starting event and
the total number of events in the system remains constant. Simulation
completes when the GVT exceeds a predetermined end time. Each
simulation thread serves a set of 128 LPs.

In the original, balanced PHOLD Model, an LP randomly picks a
destination LP from a uniform distribution to send a new message.
For our experiments, we implemented imbalanced PHOLD models
where only some portion of simulation threads receive and process
event messages in a given time period. These are called 1-2, 1-4,
1-8 and 1-16 Imbalanced PHOLD Models. For example, in the 1-2
Model, only the first half of threads (thus the first half of the total
LPs) actively communicate with each other during the first half
of the simulation. In the second half, communication shifts to the
second group. These models imitate the real-world simulations that
exhibit temporal execution locality.

2.3.2 Epidemics Simulation Model. We developed the Epidemics
Simulation Model to evaluate the GG-PDES system under a more
realistic scenario. The Epidemics Model is a location-based epidemi-
ology application where each LP resembles a single household with
a constant number of agents in it. Agents follow the SEIR Model
(Susceptible, Exposed, Infectious, Recovered) that constructs the
basis of many widely-used epidemiology frameworks [25, 28, 31]. A
simulation thread is loaded with 4096 LPs each of which containing
4 agents.

In our experiments, we evaluate lock-down effects where some
portion of the population is under curfew. Agents under the lock-

down stay in the susceptible state as they are never exposed to the
disease. We analyze scenarios where the simulated region is under
3/4 and 7/8 lock-down which forces the disease only to spread
through 0.25 and 0.125 of the entire population, respectively. When
certain LPs are under lock-down, simulation threads that handle
them do not receive and process event messages, thus they can be
de-scheduling candidates for DD-PDES and GG-PDES. The locked-
down region can shift as the simulation progresses.

2.3.3 Traffic Simulation Model. The Traffic Simulation Model from
ROSS code-base serves as another realistic application. It simulates
the movement of vehicles through a network of city intersections.
Each LP models a set of intersections and communicates with its
adjacent LPs in the four cardinal directions. The flow of vehicles
through the network is simulated by scheduling arrival, departure
and lane selection events.

The original Traffic Model produces uniform vehicular move-
ment patterns that result in similar levels of traffic activity through-
out the city. We modified it to produce more realistic traffic simu-
lations. Consequently, per LP starting events decrease as LPs are
located further away from the city center. The rate of this decrease
depends on an inverse power function in which a higher gradient
parameter leads to more centralized density distributions [3]. We
analyze the scenarios where the density gradient is set to 0.35 or
0.5. The “travel time” of each vehicle is drawn from a Burr distribu-
tion in which the parameters “c” and “k” are set to 12.4 and 0.46,
respectively [32, 33]. We load each simulation thread with 96 LPs
and the city-center LP with 24 starting events.

3 PRIOR WORK: DEMAND-DRIVEN PDES
The main principle of DD-PDES is to identify simulation threads as “active” or “inactive” based on their workloads and de-schedule
“inactive” threads from CPU cores. A thread with event messages in
its input queue is considered to be “active”. Active threads frequently
process events and contribute to the simulation progress. On the
other hand, a thread with no event messages in its input queue is
marked as “inactive” as it rarely processes an event.

An inactive thread incurs superfluous synchronization overhead.
For example, it participates in system-wide synchronization opera-
tions such as GVT computations without affecting their outcome.
Inactive threads participate in barrier calls, compete for locks used
to protect shared resources, and constantly poll input queues while
checking for incoming event messages which rarely arrive. These
all cause inactive threads to waste CPU cycles.

Figure 1 compares the traditional PDES with the original DD-
PDES and our new GG-PDES system. The traditional PDES is agnos-
tic to the thread activity as it treats all threads in the system equally
for the CPU scheduling. For example, in a 4-way simulation running
on a 4-cores machine as seen in Figure 1(a), threads with ids 3 and 4
are scheduled into available CPU cores as depicted by dotted arrows
although they are inactive. On the other hand, DD and GG-PDES
systems de-schedule inactive threads and thus prevent them from
participating in the GVT computation and polling for incoming
messages. Therefore, fewer threads compete for shared resources
and exert pressure on locks. Note that threads contain multiple LPs
and a thread is considered inactive when none of its LPs receive an
event message for a given period.
As seen in Figure 1(b), DD and GG-PDES systems also offer scalable over-subscription scenarios where a processor is overloaded with more threads than its available CPU contexts. Weak-scalability up to a large number of threads can be achieved by scheduling exclusively active threads as long as the active thread count does not exceed the total core count.

The “controller thread” of the DD-PDES system solely and explicitly manages scheduling operations. It runs on a dedicated CPU core as seen in Figure 1’s middle column. As we demonstrate later, this becomes a bottleneck for large-scale, realistic simulations.

4 GVT-GUIDED PDES

We propose a new, streamlined Demand-Driven PDES design. First, which simulation threads are active (scheduled in) at any moment. (schedules in) the inactive thread. Two global and shared arrays


4.1 GVT-Guided Scheduling

The DD-PDES implementation requires a controller thread that


4.1.3 Deactivation of Simulation Threads.

4.1.1 Architectural Overview. A simulation thread determines to be “inactive” and gets de-scheduled if its LPs have not received or sent an event message in a predefined period of time. If an LP later receives an event message, the “pseudo_controller” reactivates (schedules in) the inactive thread. Two global and shared arrays assist the deactivation and activation processes. An array of binary semaphores named “sem_locks” is used to schedule the selected threads while a boolean array named “active_threads” indicates which simulation threads are active (scheduled in) at any moment.

4.1.2 Relationship with GVT Algorithms. Asynchronous GVT algorithms are typically composed of a set of discrete phases in which simulation threads are coordinated to perform some GVT specific tasks according to the phase they belong to. For example, the Wait-Free GVT algorithm is by design composed of 5 phases in which a thread records its minimum time-stamped non-processed event in Phase A, executes a new event in Phase Send, records its new minimum in Phase B, computes a global minimum (new GVT value) based on local minimums of all threads in Phase Aware and performs the GVT housekeeping in Phase End. A simulation thread cannot advance to the next phase until all threads complete the current phase. In each GVT round, all 5 phases have to be completed by every thread for a consistent GVT computation.

Algorithm 1 Deactivation of Simulation Threads

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>function read_message_count</td>
</tr>
<tr>
<td>2</td>
<td>if input_queue[t.id].size == 0 then</td>
</tr>
<tr>
<td>3</td>
<td>zero_counter += 1</td>
</tr>
<tr>
<td>4</td>
<td>else</td>
</tr>
<tr>
<td>5</td>
<td>zero_counter = 0</td>
</tr>
<tr>
<td>6</td>
<td>active = zero_counter &gt; zero_counter_threshold</td>
</tr>
<tr>
<td>7</td>
<td>function deactivate</td>
</tr>
<tr>
<td>8</td>
<td>if active and input_queue[t.id].size == 0 then</td>
</tr>
<tr>
<td>9</td>
<td>affinity_table[affinity_table_init[t.id]] = -1</td>
</tr>
<tr>
<td>10</td>
<td>affinity_table_init[t.id] = -1</td>
</tr>
<tr>
<td>11</td>
<td>active_thread[t.id] = false</td>
</tr>
<tr>
<td>12</td>
<td>num_active_threads += 1</td>
</tr>
<tr>
<td>13</td>
<td>sem_wait(sem_locks[t.id])</td>
</tr>
<tr>
<td>14</td>
<td>num_active_threads -= 1</td>
</tr>
<tr>
<td>15</td>
<td>active_thread[t.id] = true</td>
</tr>
<tr>
<td>16</td>
<td>active = true</td>
</tr>
<tr>
<td>17</td>
<td>zero_counter = 0</td>
</tr>
</tbody>
</table>

Simulation threads invoke the “deactivate” function at the Phase End of a GVT round. A thread T deactivates based on two conditions: 1) if it checks its “active” flag as false and 2) if it checks its input queue size as 0, as shown in line 8 of Algorithm 1. T then marks its deactivation, decrements the currently active thread count and calls “sem_wait” on its semaphore to schedule out of the CPU core (lines 11 to 13). The “num_active_threads” variable is modified atomically. Lines 9 and 10 are explained in Subsection 4.2.

Eventually, T (logically one of its LPs) might receive an event message (enqueued into T’s input queue by the sender thread) which requires T to reactivate and reintegrate into the simulation. In that case, “sem_post” is called on T’s semaphore (by the “pseudo_controller”) and T schedules in and becomes active as shown in lines 14 to 17.

If T receives an event message E during its deactivation (after line 8), it deactivates normally. E is accounted for in the current
GVT round by the sender thread. T reactivates and accounts E during the next GVT round as explained in the next subsection.

4.1.4 Activation of Simulation Threads. At each GVT round, the first thread that advances to the Phase Aware is chosen as the “pseudo_controller”. In Phase Aware, the “pseudo_controller” invokes the “activate” function. If currently active threads are less than the total number of simulation threads, then the “pseudo_controller” walks through the “active_threads” array to check if an inactive thread’s input queue size is more than 0 as shown in lines 2 to 4 of Algorithm 2. A true condition implies that the corresponding inactive thread received at least one event message and should be reactivated and reintegrated into the simulation. Consequently, the “pseudo_controller” schedules the selected thread by issuing a “sem_post” on its semaphore (line 5).

Algorithm 2 Activation of Simulation Threads

1: function activate
2: if num_active_threads < num_total_threads then
3: for i := 1 to num_total_threads do
4: if active_threads[i] and input_queues[i].size > 0 then
5: sem_post(sem_locks[i]) → Activation

As the name implies, selected inactive threads are reactivated by the “pseudo_controller” at the Phase Aware of a GVT round, while active threads possibly deactivate at the Phase End. As activation and deactivation processes are coupled with GVT computation phases in which Phase Aware precedes Phase End, we ensure that the shared resources between the “pseudo_controller” and other threads, namely “active_threads” and “sem_locks” arrays can be accessed in a lock-free manner.

One might wonder why we need a controller thread at all, as inactive threads could be reactivated by threads that sent them an event message. However, this approach would create two issues: 1) concurrent access to the shared resources would need to be serialized by locks or atomic operations, 2) an inactive thread can possibly be reactivated at a moment where incorporating its event messages into the GVT computation would be infeasible.

In our experiments, we set the GVT computation frequency to 1 in 200 simulation cycles and the “zero_counter threshold” that determines how aggressively threads deactivate, to 1 in 2000 cycles. These are based on static analysis. To avoid false sharing on the “active_threads” and “sem_locks” arrays, we pad and align them to fit each cell to a cache line of our processor.

Note that unsubscribing inactive threads from the GVT computation does not violate its monotonous progress. As threads deactivate at Phase End, their pending events are already incorporated into the current GVT round. They are also re-subscribed into the GVT computation in the next round upon a single event reception.

4.2 Dynamic CPU Affinity Algorithm

The Linux process scheduler executes the Completely Fair Scheduling (CFS) algorithm and periodically performs load-balancing to migrate threads from over-utilized to the under-utilized CPU cores. This is based on the notion of scheduling domains that account for the NUMA configuration and cache hierarchy of many-core processors. However, some researchers argue that CFS and scheduling domains can fall short in delivering the optimal performance for some applications, especially the synchronization-dominated ones [4, 22].

As a consequence, a POSIX-based PDES engine can simply employ a constant CPU affinity algorithm to limit the CFS by blocking thread migrations between CPU cores in exchange for a better cache locality. Our constant CPU affinity algorithm pins threads to CPU cores in a Round-Robin fashion during the setup phase of the simulation, by using the "pthreads_setaffinity_np" function as shown in Algorithm 3.

Algorithm 3 Constant CPU Affinity

1: function set_cpu_affinity(thread_id)
2: core_id = thread_id % num_cores
3: pthreads_setaffinity_np(pthread_self(), core_id)

As the name implies, the constant affinity assignment created during the setup phase never changes. The active threads execute at CPU cores that they are initially assigned to throughout the entire simulation, even if idle cores exist as a result of thread deactivations.

When the temporal execution locality is linear which implies that active simulation threads are consecutive with respect to their threads ids, the constant CPU affinity is adequate. For example, consider the GG-PDES column in Figure 1(b): the set of currently active threads (1-4) execute on all four available CPU cores and when the locality shifts towards the next set (5-8), they still execute at 4 cores since the CPU affinity is established in a Round-Robin fashion as shown in line 2 of Algorithm 3.

However, the constant CPU affinity can cause significant performance degradations due to the load imbalance between CPU cores if the temporal execution locality is non-linear, as is very likely to be the case in real applications. In a scenario with non-linear execution locality, the set of active simulation threads is non-consecutive with respect to their thread ids. For example, if the currently active set contained threads with ids 1, 2, 7 and 8 then these 4 active threads would be forced to execute at only 2 CPU cores while the remaining 2 cores are idle. To overcome this inefficiency, we introduce the Dynamic CPU Affinity algorithm built on top of the GG-PDES system, where the affinity assignment is performed at each GVT round by the "pseudo_controller" to pin the new set of active threads to the best possible CPU cores, idled by deactivating threads.

The Dynamic CPU Affinity algorithm is executed as the last step of a GVT round after both activation and deactivation processes complete. The Dynamic CPU Affinity relies on two global arrays, “affinity_table” and “affinity_table_inv”. The former one indicates which CPU core currently contains which thread while the latter one is its inverse. For example, if affinity_table[4] is 32, then the thread with id 32 is currently active and pinned to the CPU core 4 while affinity_table_inv[32] stores 4.

In Algorithm 4, the "pseudo_controller" scans the “active_threads” array to check if an active thread T that has not yet been pinned to a CPU core exists, as shown in lines 3 to 6. To find an idle core for T, the "pseudo_controller" then scans the “affinity_table” followed by “affinity_table_inv” (lines 7 and 8). When an idle core is found, tables are updated accordingly (lines 9 and 10). T is pinned using the system call “sched_setaffinity” (line 11) and the loop breaks to possibly pin the next active thread to the next idle core (line 12). The core_id counter is declared at the most outer scope (line 2) to short-circuit the inner for loop (line
Algorithm 4 Dynamic CPU Affinity

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>function set_cpu_affinity</td>
</tr>
<tr>
<td>2.</td>
<td>core_id ← 0</td>
</tr>
<tr>
<td>3.</td>
<td>for t_id ← 0 to num_total_threads do</td>
</tr>
<tr>
<td>4.</td>
<td>if active_threads[t_id] then</td>
</tr>
<tr>
<td>5.</td>
<td>if affinity_table_in[t_id] &gt;= 0 then</td>
</tr>
<tr>
<td>6.</td>
<td>continue</td>
</tr>
<tr>
<td>7.</td>
<td>for core_id to num_cores do</td>
</tr>
<tr>
<td>8.</td>
<td>if affinity_table[core_id] == t_id then</td>
</tr>
<tr>
<td>9.</td>
<td>affinity_table[core_id] ← t_id</td>
</tr>
<tr>
<td>10.</td>
<td>if core_id has Thread t_id then</td>
</tr>
<tr>
<td>11.</td>
<td>sched_setaffinity(t_id, core_id)</td>
</tr>
<tr>
<td>12.</td>
<td>break</td>
</tr>
</tbody>
</table>

7). A deactivating thread reset associated indexes at both tables as shown in lines 9 and 10 of Algorithm 1.

The Dynamic CPU Affinity is Simultaneous Multi-threading (SMT)-aware. This is important for modern many-core processors, where each CPU core usually supports multiple hardware threads. When an idle CPU core is searched (lines 7 and 8), the “pseudo_controller” favors cores with the least amount of active hardware threads. For example, if there are 4 currently active simulation threads, they will be pinned to 4 different CPU cores, ensuring none of them shares its core with another active hardware thread. Our current implementation does not account for asymmetrical architectures or NUMA capabilities.

5 EXPERIMENTAL FRAMEWORK

We conduct our experiments on Intel’s second-generation Xeon Phi processor, the Knights Landing (KNL). Our KNL Model 7230 is used as a standalone processor. It has 64 CPU cores with 4-way Simultaneous Multi-threading (SMT), in total providing 256 hardware threads. It includes a 16 GB on-package MCDRAM and 96 GB DDR4 main memory. Our system runs CentOS 7.2 with Kernel version 3.10.0 and GCC version 8.3.1. We enable 02 optimization for all our experiments.

We use weak scaling for our scalability tests. The committed event rate is used to report the performance of a system. The committed event rate is the number of events committed over the total number of events executed per second in wall-clock time. We also utilize the PAPI tool [34] and rdtsc (Read Time-Stamp Counter) x86 assembly instructions for our profiling purposes.

6 RESULTS AND DISCUSSION

GG-PDES system comes in two flavors: GG-PDES-Sync and GG-PDES-Async. We evaluate these against original DD-PDES systems: DD-PDES-Sync and DD-PDES-Async and two baseline implementations: Baseline-Sync and Baseline-Async. Synchronous and asynchronous (-Sync and -Async) systems are supported by Barrier GVT and Wait-Free GVT algorithms, respectively. Baseline systems do not perform explicit thread scheduling as DD and GG-PDES do and simply rely on the default, Completely Fair Scheduling (CFS) algorithm of our CentOS kernel. The scheduling of simulation threads (active or inactive) is exclusively managed by the CFS.

6.1 Balanced PHOLD Model

We start our evaluation with a balanced simulation model to quantify the overhead of DD and GG-PDES systems that are primarily designed to exploit the temporal execution locality typically present in imbalanced simulation models. As seen in Figure 2, GG-PDES performs competitively in a balanced model with no execution locality. In a 256-way simulation, GG-PDES-Async outperformed by Baseline-Sync by only 4.3% while GG-PDES-Sync outperforms Baseline-Sync by 1.5%. Our KNL model can run 256 simulation threads (POSIX) simultaneously as it contains 64 CPU cores, each with 4-way SMT.

Figure 2: Committed Event Rates under Balanced PHOLD Model

The small performance degradation of GG-PDES-Async compared to Baseline-Async indicates the overhead of checks that are performed for deactivation conditions is minimal and our implementation is quite light-weight. For example, in a 256-way simulation, the average CPU time spent for a GVT computation round (accumulated among threads) is 0.92 and 0.95 seconds for Baseline-Sync and GG-PDES-Async, respectively. As all the activation and deactivation logic is executed within GVT functions, GVT timings are a good indicator of GG-PDES overhead. The reason that GG-PDES-Async outperforms Baseline-Sync stems from its lighter, customised barrier functions.

6.2 Moderately Imbalanced PHOLD Models

In 1-2 Imbalanced PHOLD Model, only half of the simulation threads are active at a time as their LPs communicate exclusively. The group of active threads shifts as the simulation progresses. This is generally referred to as temporal execution locality. Similarly, in the 1-4 Imbalanced Model, only a quarter of simulation threads are active at a time. As shown in Figures 3(a) and 3(b), GG-PDES-Async outperforms Baseline-Sync by 10% and 17% at 256-way simulations, respectively.

As only half of the simulation threads are active in 1-2 Imbalanced Model, we can over-subscribe our processor and scale up to 512 threads. Similarly, we over-subscribe 1-4 Imbalanced Model with 1024 simulation threads. GG-PDES-Async outperforms Baseline-Sync by 5.4% and 14% at 512-way simulation of 1-2 and 1024-way simulation of 1-4 Imbalanced Models, respectively. DD-PDES implementations perform competitively until we over-subscribe. As the simulation scales up, the centralized design built around the controller thread becomes a bottleneck and results in the performance of DD-PDES plummeting.

As seen in Figure 3, Baseline-Sync significantly outperforms Baseline-Async. Because the pthread_barrier function deployed in Barrier GVT algorithm of Baseline-Sync schedules out inactive
threads while all threads (active or inactive) are kept scheduled throughout the simulation with Baseline-Async. At each GVT round, inactive threads which arrive at the barrier function earlier are scheduled out until all threads synchronize at the barrier. Unlike Baseline-Sync, GG-PDES schedules out inactive threads systematically, based on their input queues sizes and keeps them scheduled out as long as needed (until they receive a message and become active again).

GG-PDES systems reduce the number of threads involved in the system-wide synchronization operations such as the GVT computation and accelerate their speed. The average CPU time spent for a GVT round in a 512-way simulation of 1-2 Imbalanced Model, is 3.88 and 3.15 seconds for GG-PDES-Async and GG-PDES-Sync, respectively. However, these numbers are 137.33 and 33.07 for Baseline-Async and Baseline-Sync. Similarly, in 1024-way simulation of 1-4 Imbalanced Model, an average GVT round lasts 6.18 and 3.78 seconds for GG-PDES-Async and GG-PDES-Sync, respectively while for Baseline-Async and Baseline-Sync, these numbers are 574.62 and 104.7 seconds.

We also analyze the impact of GG-PDES on CPU utilization. Using the PAPI profiler [34], we compute the total number of instructions executed among CPU cores during the core simulation loop of ROSS. Compared to baseline implementations, GG-PDES-Async completes a simulation by executing the lowest number of instructions as it dispenses inactive threads’ instructions. In 512-way simulation of 1-2 Imbalanced Model, GG-PDES-Async and Baseline-Sync execute around 0.16 and 0.31 trillion instructions, respectively. Similarly, in 1024-way simulation of 1-4 Imbalanced Model, GG-PDES-Async and Baseline-Sync execute around 0.08 and 0.29 trillion instructions, respectively.

One might wonder why a practitioner would develop a model with temporal execution locality and not partition it well for parallel execution. For example, the group of active LPs could be evenly distributed across threads to make them remain more-or-less active throughout the simulation. We argue that this would require a practitioner to spend extra time and effort to characterize and address their model’s execution locality which can highly vary based on model parameters or dynamically change as is the case in our experiments. On the other hand, numerous dynamic partitioning techniques are proposed which make different trade-offs to sustain a balanced execution although up to our knowledge, none accelerates the system-wide synchronization operations or introduces over-subscription scenarios.

6.3 Highly Imbalanced PHOLD Models

Highly Imbalanced Models exhibit a higher temporal execution locality compared to Moderately Imbalanced Models as fewer threads are active at a time. For example, in 1-8 and 1-16 Imbalanced Models, only 12.5% and 6.25% of simulation threads, and subsequently their LPs, actively receive and process event messages, respectively.

In 256-way simulations, GG-PDES-Async outperforms Baseline-Sync by 8.5% and 11% as seen in Figures 4(a) and 4(b), respectively. When we over-subscribe, these numbers rise to 18% and 44% in a 2048-way simulation of 1-8 and 4096-way simulation of 1-16 Imbalanced Models, respectively. These performance improvements are proportional to the amount of temporal execution locality present in simulation models. As observed before, DD-PDES implementations perform competitively until the simulation scales up to 512 threads.
Figure 5: Committed Event Rates under Epidemics Simulation Models

GG-PDES accelerates the system-wide synchronization by eliminating the unnecessary involvement of a large number of threads. For example, in a 2048-way simulation of 1-8 Imbalanced Model, GG-PDES-Async spends an average CPU time of 9.9 seconds for a GVT computation while Baseline-Sync spends 289.8 seconds. Similarly, in a 4096-way simulation of 1-16 Imbalanced Model, GG-PDES-Async completes a GVT round in 19.3 seconds on average while Baseline-Sync does so in 805.6 seconds. Also, GG-PDES-Async executes around 0.044 and 0.27 trillion instructions while Baseline-Sync executes 0.31 and 0.37 in 512-way simulations of 1-8 and 1-16 Imbalanced Models, respectively.

6.4 Epidemics Simulation Model

As Baseline-Async, DD-PDES-Sync and GG-PDES-Sync generally underperformed in PHOLD experiments, we evaluate only Baseline-Sync, DD-PDES-Async and GG-PDES-Async, denoted as Baseline, DD-PDES and GG-PDES to emphasize our main findings.

Figures 5(a) and 5(b) present results of Epidemics Simulation Models with 3/4 and 7/8 lock-down rates, respectively. We observe a pattern similar to PHOLD experiments where GG-PDES achieves 22% and 29% performance improvements over Baseline in 256-way simulations as seen in Figures 5(a) and 5(b), respectively. When we over-subscribe, these numbers become 13% in 1024-way simulation of Figure 5(a) and 19% in 2048-way of Figure 5(b). The performance gap widens as models incur a higher temporal execution locality (higher lock-down rate). On average, the CPU time spent for a GVT computation is 52.4 seconds in GG-PDES and 324.6 seconds in Baseline under the 2048-way Epidemics Simulation with a 7/8 lock-down rate.

Figure 6: Committed Event Rates under Traffic Simulation Models

6.5 Traffic Simulation Model

Figures 6(a) and 6(b) present results of Traffic Simulation Models with density gradients of 0.35 and 0.5, respectively. As the temporal execution locality is limited in these models, performance advantages of GG-PDES degrade. While it slightly underperforms Baseline in 256-way simulations, a speedup in the range of 24% and 27% can be achieved in 512-way simulations as seen in Figures 6(a) and 6(b), respectively. For the 512-way simulation of 0.5 gradient, the average CPU time spent for a GVT computation is 7.9 seconds in GG-PDES and 11.5 seconds in Baseline.

The performance degradation at larger scales (1024 and more simulation threads) stems from an increase in the number of rolled-back events. For example, in the 2048-way simulation of 0.5 gradient, GG-PDES processes 540.4 million events, of which 360.3 million are rolled-back. Meanwhile, Baseline processes 562.4 million events, of which 416.0 million are rolled-back. DD-PDES processes 1.18 billion events, of which 1.03 billion are rolled-back.

6.6 Dynamic CPU Affinity

The Constant CPU Affinity algorithm pins simulation threads to CPU cores in a Round-Robin fashion and offers performance improvements due to a better cache locality. The No CPU Affinity algorithm simply does not perform any CPU pinning and allows the Linux process scheduler to migrate threads between CPU cores for a balanced execution. Our key observation is that the Constant CPU Affinity is beneficial as long as the simulation model exhibits linear execution locality where the set of active threads are consecutive with respect to their thread ids.

Figure 7 presents the performance results of GG-PDES-Async under three different CPU affinity algorithms: No CPU Affinity,
Affinity does not pin threads to cores, the Linux process scheduler supports the Dynamic CPU Affinity which pins active threads dynamically, based on currently active threads and idle CPU cores. As opposed to the Constant CPU Affinity, the pinning schema (Round-Robin) never alters based on idle cores, the Constant CPU Affinity pins more than one thread to the same CPU core, although idle CPU cores exist as a result of thread deactivations. As the pinning schema (Round-Robin) never alters based on idle cores, the Constant CPU Affinity causes significant load imbalance between cores and performance can significantly degrade, as seen in Figure 7(b). As the No CPU Affinity does not pin threads to cores, the Linux process scheduler can select an idle CPU core to run an active thread which prevents the load imbalance up to some degree.

GG-PDES supports the Dynamic CPU Affinity which pins active threads dynamically, based on currently active threads and idle CPU cores. As opposed to the Constant CPU Affinity, the pinning schema is mutable and varies to achieve a good load balance between CPU cores. The Dynamic CPU Affinity achieves up to 33% and 15X performance improvements over No CPU Affinity and Constant CPU Affinity algorithms respectively, if the simulation model exhibits non-linear execution locality as it is the case in Figure 7(b).

Although the Dynamic CPU Affinity is primarily designed to exploit the non-linear execution locality, it also performs competitively when the execution locality is linear. It achieves up to 35% performance improvements over the No CPU Affinity and degrades only by 0.5% compared to the Constant CPU Affinity, as seen in Figure 7(a). The small degradation is the cost of extra checks performed by the "pseudo_controller". The largest scale simulation we conduct includes 4096 simulation threads and results in the Dynamic CPU Affinity algorithm’s memory footprint to reach 17 KB. As our KNL processor contains 32 L2 caches of size 1 MB each, this memory footprint is insignificant.

7 RELATED WORK

Dynamic object migration and load balancing in the SPEEDES simulation framework are demonstrated in [39]. Researchers in [2] present multiple dynamic model partitioning algorithms and compare them with static partitioning. The work of [7] proposed a more light-weight approach that migrates workloads only between neighboring LPs. A "share-everything" system proposed in [17] enables LPs to move from over-utilized threads to vacant ones. The work of [23] introduced an adaptive load-balancing algorithm by leveraging the GVT computation. A detailed performance characterization of load balancing algorithms is presented in optimistic simulations [36] and on many-core platforms [29]. These load-balancing algorithms are multi-objective: they aim to sustain a balanced workload distribution while the communication between co-executing threads should also be confined. GG-PDES provides a more straightforward approach where inactive threads are simply descheduled from CPU cores. This also accelerates system-wide synchronization operations as fewer threads compete for synchronization primitives and offers over-subscription possibilities, unlike load-balancing algorithms.

A comprehensive performance characterization of GVT algorithms is presented in [11]. Authors in [18] propose a non-blocking, wait-free GVT algorithm. An adaptive algorithm that dynamically determines the optimum GVT computation frequency is proposed in [24]. A similar idea is also applied to a barrier-based GVT algorithm in [37]. Perumalla et al. demonstrate scalability up to 200,000 cores through novel GVT algorithms [27]. The work of [9] proposes a semi-synchronous GVT algorithm to throttle the optimism level based on rollback rates. A recent proposal dynamically switches between the optimistic and conservative processing based on the localization of event messages [8] in a PDES-based GPGPU simulation [1]. A comprehensive performance characterization of PDES systems on a KNL processor is presented in [38].

Over-subscription scenarios interest researchers from a wide range of domains. Authors in [16] present a comprehensive analysis of over-subscription scenarios for various parallel programming models running on multi-socket, many-core processors. The work of [40] over-subscribes compute nodes in a distributed system by relocating processes based on their resource consumption characteristics. Huang et al. [15] present a novel MPI implementation where several virtual MPI processes can be mapped to a single physical core. The work of [6] evaluates the resource sharing on SMT processors using OpenMP programs and reports the over-subscription on SMT cores can degrade the overall performance.

Figure 7(b). Non-Linear Execution Locality

Figure 7: Committed Event Rate of GG-PDES-Async with Different CPU Affinity Algorithms under PHOLD Models
8 CONCLUDING REMARKS

In this paper, we identified several performance bottlenecks in the recently proposed Demand-Driven PDES system, which de-schedules simulation threads if they have not been recently sending or receiving events. To address these inefficiencies, we proposed GVT-Guided PDES that introduces a streamlined, lock-free scheduling design and a dynamic CPU affinity algorithm. Together, these optimizations result in substantial performance improvements over the original DD-PDES system.

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


