Hybrid Scheduling for Event-driven Simulation over Heterogeneous Computers

Bilel Ben Romdhanne
Mobile Communication
Eurecom
benromdh@eurecom.fr

Mohamed Said Mosli Bouksiaa
Mobile Communication
Eurecom
mosli@eurecom.fr

Navid Nikaein
Mobile Communication
Eurecom
nikaeinn@eurecom.fr

Christian Bonnet
Mobile Comminication
Eurecom
bonnet@eurecom.fr

ABSTRACT

In this work we propose a new scheduling approach designed from scratch to maximize heterogeneous computers usage and the event processing flow at the same time. The scheduler is built based on three fundamental concepts which introduces a new vision of discrete event simulation: 1) events are clustered according to their potential time parallelism on one hand and to their potential process and data similarity on the other hand. 2) events meta-data is enhanced with additional descriptor which simplifies and accelerates the scheduling decision. 3) the simulation is hybrid time-event driven rather than time- or event-driven. The concretization of our approach is denoted the H-scheduler which uses several processes to manage the event flow. Furthermore we propose a dynamic scheduling optimization which aims to further maximize the event flow. The combination of those features allows the H-scheduler to provide the highest efficiency rate compared to the majority of GPU and CPU schedulers. In particular it goes beyond the default Cunetsim Scheduler by 90% in average while it keeps a significant lead on existing simulators.

Categories and Subject Descriptors

I.6.0 [**Computing Methodologies**]: SIMULATION AND MODELING—*General*; C.4 [**Computer Systems Organization**]: PERFORMANCE OF SYSTEMS

Keywords

PADS, PDES, Large scale simulation, scheduling methodology, GPGPU, Heterogeneous computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSIM-PADS'13, May 19–22, 2013, MontrÃl'al, QuÃl'bec, Canada.
Copyright 2013 ACM 978-1-4503-1920-1/13/05 ...$15.00.

1. INTRODUCTION

Discrete event simulation (DES) is a tool used to model, analyze, and evaluate large and complex systems requiring continuous internal and external interactions, where formal analysis is difficult or non-deterministic. However, the efficiency and scalability of DES remains a challenge due to emerging heterogeneous computing node architecture. In particular, scheduling the execution of future events while maintaining a continuous load under large-scale conditions increases the scheduling cost, until becoming the bottleneck [12]. Most popular scheduling approaches rely on a centralized event scheduling model optimized mainly for the homogeneous computing node architecture. Such a model remains limited and does not exploit the full modern hardware potential. Hence, the parallel and distributed scheduling approaches used for parallel discrete event simulation (PDES) re-emerge as a candidate to increase the event generation rate over heterogeneous computing node architectures [25]. The objective is to exploit a multitude of parallel and interactive processors unified at the level of event scheduler to cooperate with each other. Examples include multi-core CPUs, multi-GPUs, multi processor system-on-chip, and accelerated processing unit introduced by INTEL.

Most of those architectures seem promising, but their ecosystems in some cases are either not fully developed or conflicting [5]. Benefits of GPU-enabled supercomputer have been highlighted in [21], where authors suggest revisiting and expanding the vision on DES. Nevertheless, most of the recent attempts assume the backward compatibility with the sequential scheduling concept [16, 27]. Such a methodology presents a conceptual weakness since it considers a multi-core computing node as a simple extension of a mono-core one. Furthermore, to remain backward compatible, the expected gain will be significantly reduced when compared to a dedicated software design which considers the parallel specificity of the current hardware as well as the communication latency [1].

In this work, we introduce a new parallel event scheduler for a heterogeneous computing node architecture, denoted as Hybrid scheduler (H-scheduler). The H-scheduler is designed to dynamically allocate events to the available computing resources while keeping the event rate and load stable. This is achieved as the scheduler is aware of the heap of processors and their capabilities and has a constant access to

their instantaneous loads and execution time. The scheduler operates on all the available computing resources within the same addressable memory space. To increase the efficiency of the scheduler, the events are associated with metadata and organized in a 3-dimensional data structure. Furthermore, events are considered as different flows from the scheduler to each computing resources. The H-scheduler applies the conservative scheduling policy to avoid the overhead generated by the recovery mechanism and the state vector when considering optimistic policy in parallel and heterogeneous settings.

The H-scheduler is composed of four main processes: event dispatcher, event injector, GPU-scheduler, and CPU-scheduler, where event are flowing. The dispatcher fetches the newly generated events from different queues and adds them to a corresponding position within a 3-dimensional data structure optimized for the parallel execution while the injector directs a group of parallel events to the most adequate sub-scheduler based on the received feedback information. Each sub-scheduler is tailored and optimized for a specific hardware in order to maximize the activity rate of the corresponding computing resources.

Several optimizations are proposed to accelerate the scheduling decision as the bottleneck may change over time. To demonstrate, we adopt rapid and advanced scheduling policies for both the dispatcher and the injector processes, and enable switching dynamically between them based on feedback information. Comparative assessments have been demonstrated that the performance gain can be increased by a factor of 2 compared to centralized and conservative schedulers.

The reminder of this paper is organized as follows. Section 2 provides a related work on the scheduling in the event-driven simulation. Section 3 presents the H-scheduler in detail. The performance assessments and analysis are described in Section 4 followed by a discussion in Section 5. Finally, the conclusion is presented in Section 6.

2. RELATED WORK

Improving the efficiency and scalability of DES remains a challenging issue for the modern modeling approaches that require complex and sophisticated system representation. In such a context, PDES is commonly used as a scalable and efficient solution when compared to sequential approaches [7]. PDES relies on the partitioning of the model over several logical processes (LP)s collaborating with each other to perform the whole simulation [22, 13]. However respecting the simulation correctness while dealing with parallel execution makes event management extremely expensive. This is also acknowledged as one of the critical limitations of large parallel simulations, especially when dealing with heterogeneous resources, and raises two issues: data representation and event scheduling [20].

To store future events, most of the PDES use a sorted data structure. In the literature, the efficiency of central data structures was largely studied for both sequential and parallel execution, e.g. central event list (CEL) [23]. Nevertheless, under large parallelism conditions, such a data structure may present a bottleneck. Authors in [7] address the efficiency of three CEL implementations, namely the heap, the splay tree and the calendar, and they conclude that the performance of the CEL concept remains mitigated when thousands of concurrent processes access that structure. Therefore, the CEL implementation needs to be parallelized to cope with the parallel architecture of heterogeneous computing resources. The concurrent priority queue [24, 9] is a relevant solution to access and manage the CEL in parallel. An event list or message queue is usually distributed to each logical process in a PDES with its own local clock. The concurrent insertion and deletion of the priority queue, by involving mutual exclusion or atomic functions, leads to the improvement of the overall performance using a global event list [23]. In the same sense, Chen *et al.* propose a distributed queue which considers multi-core CPUs [6]. However, the above mentioned mechanisms, mutual exclusion and concurrent priority queue, are target-dependent and thus could not be directly applied to the GPU target. A different point of view was proposed by Park *et al.* [17], which relies on a hybrid time-event driven simulation based on a GPU-oriented CEL concept that uses a linked list implementation. Despite the fact that this approach has been developed with the goal of improving the GPU-based simulation, the overhead of managing a large number of parallel event remains an open issue due to a limited number of concurrent access to the same physical memory.

In the DES and PDES, a large portion of the overall simulation time is used for the event scheduling [18]. Specifically to PDES, the efficiency of the scheduler depends also on the synchronization method [14]. The conservative approaches prohibit out-of-order event execution, which in most cases is based on the lookahead concept to preserve the causality rule [11]. Parallel event scheduling is separately studied for a multi-core CPU target [27, 8] and GPUs [17]. Both approaches use a central event queue and several independent threads to fetch the next event from the queue. The multi-threading approach is also applied to reduce the scheduling cost and increase the simulation efficiency but only for a limited number of cores (4 and 8 respectively). Authors in [16] propose a dedicated GPU scheduling based on the SIMD programming model where the event queue is splitted into several sub-queues to avoid central bottlenecks. All the above mentioned approaches regard a GPU core as a CPU core, which in turn reduces the achievable gain.

The optimistic approach allows an out-of-order execution while ensuring the simulation correctness. Several optimizations were introduced recently to increase the efficiency of optimistic parallel simulations over multi-core CPUs while keeping a reasonable backward compatibility with standard software architecture [26, 15, 6]. Although such an approach increases the general efficiency by relaxing a substantial limitation of the conservative approach, it introduces memory overhead related to the state-vector saving mechanism, which becomes critical when targeting very large logical process.

In contrast with previous works, the H-scheduler design consider extremely large LPs while maximizing the simulation efficiency over heterogeneous computing architecture. In particular it outperforms prior works by considering events as flow and detecting the system bottleneck on the fly to adjust the behavior of the scheduler dynamically.

3. THE HYBRID SCHEDULER

To operate on a heterogeneous computing node, the hybrid scheduler relies on four fundamental concepts: (A) event descriptor, which associates metadata to an event when it is generated, (B) event structure, which organizes the events in an arraylist (AL) - namely cloned independent events (CIE)

if the events differ only in data, independent foreign events (IFE) if events differ in both algorithm and data, and time interval - and (C) event flow, which considers the simulation process as a set of interconnected sub-processes producing and consuming events where the event rate stability across multiple computing resources has to be maintained. In the following paragraphs, we further explain each concept.

The event descriptor extends the traditional event metadata, namely timestamps, id, in/out data, to support additional information used to reduce the parallel event scheduling cost. In particular, it includes (i) event dependency information, (ii) event execution timestamp, (iii) I/O data access, (iv) event structure information, and (v) execution targets. Event dependency defines if an event has one or multiple dependencies (needs current output as input) that fall within the same time interval [3]. The event execution timestamp identifies which events can be scheduled in parallel for a given timestamp and is calculated based on the current timestamp and the safety lookahead. The I/O data access defines the permissions given to an event to read and/or write a shared memory area. The event structure information identifies an event as a CIE or a IFE for the given timestamp. Finally, the execution target defines where an event could be executed, CPU, GPU or both. It has to be mentioned that the scheduler can not infer the target without using system callback which is a costly procedure. Therefore including that information on the event descriptor accelerate the scheduling decision.

The event structure expands how the events are represented so as to increase the parallelism in a heterogeneous computing node architecture. For this purpose, events are dispatched over a 3-dimensional arraylist, where each element of the array represents the timestamp and the associated list represents the parallel event sets for a given timestamp interval. The parallel event set is composed of the CIE (SIMD-like) and IFE the (MIMD-like). It has to be mentioned that, the CIE are considered by the scheduler as an event set (ES) and processed by the scheduler as a unique entry while the IFE are considered as a heap of events and processed by the scheduler as multiple entries. If an IFE event has one or multiple dependencies that fall within the same time interval, they will be executed sequentially to preserve the correctness.

The event flow considers the simulation as a dynamic system where events are flowing between the producers and consumers. Depending on the simulation characteristics and the available resources, system bottleneck may change over time. Therefore, the event rate stability has to be dynamically maintained so as to maximize the simulation efficiency. Consequently, feedback information from each computing resource is needed to control the event rate through the scheduling. In the following sections, we detail the the scheduler design and algorithms.

3.1 Model and components

To perform an efficient parallel event scheduling in large scale conditions over a heterogeneous computing node, the scheduler is designed as a composition of several processes. It includes the future event list (FEL), the dispatcher, arraylists (AL), the injector, the GPU-scheduler, the CPU-scheduler, and feedback as shown in the Figure 2. The FEL is a standard FIFO providing reliable status flag used to collect and manage the produced events. There is one dedicated FEL per computing resources to gather the generated events and one FEL for all the incoming raw events. The dispatcher is the front-end of the scheduler, it first reads the events from the FEL following a given scheduling policy (e.g. weighted round robin). Then, it adds events into the global 3-dimensional AL data structure based on the event descriptor to ensure the simulation correctness. The injector proceeds on the per-interval basis and dynamically determines the target computing resource, i.e. GPU or CPU, as well as the subset of events to be allocated based on the event descriptor, received feedback information and the target capabilities.

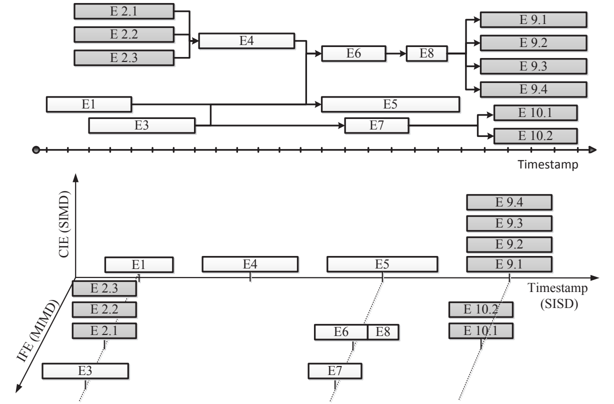

Figure 1: Event Structure

As a result, the subset of allocated events will be pushed to the local AL of a target. Please note that the injector starts the next interval only when all the events associated to the current interval are executed. Both the GPU and CPU scheduler receive events on a dedicated local AL buffer, where events are organized in the CIE and the IFE sets (see Figure 1). The dedicated GPU scheduler maps each entry of the AL to an asynchronous and non-blocking GPGPU calls. On the other hand, the dedicated CPU-scheduler will make use of multi-processing/threading technology (e.g. openMP). Upon the execution of an event, a new event might be generated (similar to producer-consumer processes) and pushed into the dedicated FEL. To dynamically adjust the event flow, each target sends feedback information about the instantaneous load and the execution time per event.

3.2 Scheduling Algorithms

In this part, we detail the different algorithms on which the H-scheduler is based, namely the advanced algorithm, the rapid algorithm and the hybrid algorithm.

3.2.1 Advanced algorithm

The advanced algorithm aims to thoroughly select the most adequate target for each event. The events flow starts from the FEL where all events are firstly inserted. The dispatcher is the first process which handles the event in order to push it on the correct place of the main AL. Based on the event timestamps, the dispatcher determines the correct sub-list where the event must be inserted. Then, it starts resolving its dependency using the event descriptor. If the

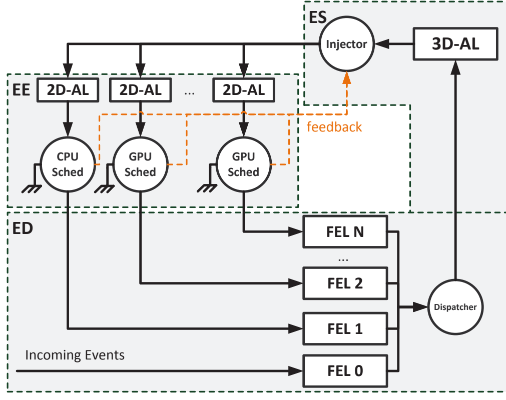

Figure 2: Event scheduler model

dispatcher detects a dependency between two events it has three choices: splitting the interval into two new ones, each of which includes independent events while respecting the timestamps correctness, creating a merged event including both or transforming the sub-list to a sequential one. Algo 1 presents the pseudo code of the dispatcher. There are two issues related to that process: detecting dependency and resolving it. To detect the dependency of two events, the dispatcher relies on their descriptors as follows: first it explores their explicit dependency descriptors. Second, it computes their durations, thus if both events are concurrent then they will be considered as independent. Third, if both events do not use the same data, the dispatcher concludes they are independent (this features must be explicitly enabled by the user). To resolve the dependency, the most adequate choice is to split the interval. However, if such a procedure induces a conflicting situation, the dispatcher creates a merged event. Finally, if dealing with the interval becomes complicated due to the large number of included events, then the sub-list will be transformed into a sorted list. The injector processes the AL sequentially, sub-list

```
for e ∈ FEL do
    if e.timestamps ∉ existing interval then
        createNewInterval(I, e.timestamps);
        insertEvent(I,e);
    end
    else
        for e1 ∈ I do
            if dependency(e1,e) then
                resolve(e1,e);
            end
        end
    end
end
```

Algorithm 1: Pseudo code of the dispatcher.

by sub-list. If the sub-list is sequential, then all events will be forwarded to one target. Otherwise, the injector considers each event individually based on the following routine: if the event is mono-compliant (CPU or GPU), then it is switched to the adequate sub-scheduler. If there are several instances of the target (several GPU or CPU scheduler then it will be switched to that having the lowest load currently. In what concerns grouped entries, the injector analyses the parallelism information to determine the target. If the number of parallel instances is reduced then the CPU is the most adequate target, and if that number is extremely large, the chosen target is the GPU. The boundaries of this decision are a function of the number of CPU & GPU cores. In the beginning of the simulation we use two arbitrary intervals where the decision is deterministic: $[1, 2 * core * 3 + 1] \Rightarrow$ CPU and $[200, +\infty] \Rightarrow$ GPU. Nevertheless, if the parallelism size is intermediate (number of instances greater than CPU's interval's upper boundary and lower than GPU's interval's lower boundary), then the injector inspects the load of each target and chooses the available one. Finally, it ensures the synchronization of all secondary ALs using a synchronization checkpoint at the end of each interval to maintain the correctness of the simulation.

```
for I ∈ MAL do
    for e ∈ I do
        if e.target=CPU then
            schedule(e,CPU);
        end
        else
            if e.target=GPU then
                schedule(e,GPU);
            end
            else
                if e.instance ∈ [1, NCPU] then
                    schedule(e,CPU);
                end
                else
                    if e.instance ∈ [NGPU, +∞] then
                        schedule(e,GPU);
                    end
                    else
                        balancedschedule(e,GPU,CPU);
                    end
                end
            end
        end
        Synchronize(I);
    end
end
```

Algorithm 2: Pseudo code of the injector.

The CPU-scheduler ensures event execution over available CPU cores. At the initialization phase, the CPU-scheduler starts by discovering available resources (asking the hardware and reading the configuration file); then it creates as many execution threads as available cores. Second, it feeds execution threads with events as soon as possible, without dependency control. Therefore, events of one sub-list are expected to be executed in parallel over available resources. Since the H-scheduler respects a conservative scheduling approach, it does not execute events of different sub-lists concurrently. The CPU-scheduler notifies the injector when finishing an interval and waits for its permission to consider the next one. The GPU-scheduler is slightly different, since it relies on a hybrid software-hardware scheduling mechanism. At the software level, sub-list includes always grouped entries that it translates to a CUDA call with pre-

defined generic parameters. The CUDA driver ensures next steps, including generating threads and sending them to the GPU. At the hardware level, the embedded GPU GigaThread scheduler first distributes event thread blocks to various SMs, and second assigns each individual thread to an SP inside the corresponding SM.

3.2.2 *Rapid algorithm*

The rapid algorithm is a simplified version of the advanced one which aims to minimize the decision cost. It concerns particularly the dispatcher and the injector. It relies on a major simplification of the bottleneck of each process using a deterministic model.

As for the dispatcher, the most expensive routine is the dependency resolution; in particular, splitting an interval in two independent ones requires an expensive modification of the main AL structure. The rapid algorithm proposes to use always combined events if the dispatcher detects any dependency into one interval. We note however, that the dependency detection routine can not be simplified since it affects the correctness of the simulation.

Concerning the injector, the most expensive routine is the identification of the most suitable target. The rapid algorithm reduces its complexity by extending the borders of decision intervals for both CPU and GPU: the CPU interval becomes $[1, N]$, and the GPU one becomes $[N+1, +\infty]$, where N is a tuning parameter. The second critical routine is how to determine the most adequate target if we have multiple instances (multiple CPUs or GPUs). In that case, rather than evaluating resources loads, the injector uses a round robin assignment mechanism which aims to ensures a minimal load balancing based on the number of events.

3.2.3 *Hybrid Algorithm*

The hybrid algorithm aims to ensure the maximal stability for the system. It relies on two mechanisms: algorithm switching and the parameter recalibration. The switching mechanism changes the operating algorithm for both injector and dispatcher processes based on a bottleneck detection approach. Each process has one inbuffer which includes one filling rate attribute. The inbuffer of the injector is the main AL, and that of the dispatcher is the FEL. For each process, if the inbuffer is full, the hybrid mechanism supposes that the consumer process is the bottleneck and acts as follows: if the filling rate is between empty (E) and almost empty (AE) then the selected algorithm is the advanced. If that rate is between AE and almost full (AF) then the selected algorithm remains the advanced one, but the recalibration frequency is increased. Finally, if the inbuffer filling rate is between AF and full (F) then the selected algorithm is the rapid one. The switching decision occurs between two intervals and cannot be achieved during the execution of an interval.

The recalibration mechanism computes continuously the values of three parameters which define the behavior of both algorithms: the NCPU & NGPU of the advanced algorithm and the N of the rapid one. It maintains a statistical table which includes the average execution time of event sets in different targets. Figure 3 presents an example of a statistical table, where the recalibration process can assert that $NCPU = 24$, $NGPU = 192$ and $N = 48$. To compute the average execution time of a specific event set size, the recalibration mechanism considers the last M samples (which is a tuning parameter defined by the user), therefore the statistical table copes with the evolution of the simulation on one hand and the hardware characteristic on the other hand.

Target\ size	1	...	...	24	36	42	48	54	...	192	288	384
CPU	0.2	...	...	0.8	1	1	1	1.5	...	6.4	9.6	...
GPU	1	1	1	1	1	1	1	1	1	1	2	...

Figure 3: Statistical table used for recalibration

4. PERFORMANCE EVALUATION

To implement the H-scheduler, we rely on the network simulation framework Cunetsim [3, 4]. Both the H-scheduler and Cunetsim rely on five parallelization frameworks, namely CUDA, OpenMP, MPI, the thrust data management API [2], and the PGI development Kit [28], which are briefly explained below.

1. The Compute Unified Device Architecture (CUDA) is a software parallel computing platform and a programming model created by NVIDIA. In what concerns this work, the last CUDA release provides two main features: atomic operations and the GPUDirect technology which accelerates the communication between the GPU and the different components of the computer.

2. The Open Multiprocessing (OpenMP) is an API that supports multiprocessing in a shared memory context. We rely on the OpenMP to provide a compliant version with multi-core CPUs as explained later.

3. The Message Passing Interface (MPI) is a standardized and portable message-passing system designed to supply programmers with a standard for distributed programming.We rely on MPI to ensure the communication and the synchronization between different ELPs of the system.

4. Thrust is a parallel algorithm library which imitates the C++ Standard Template Library (STL). Thrust's interface enables performance portability between GPUs and multicore CPUs. Interoperability with established technologies (such as CUDA, TBB and OpenMP) facilitates integration with existing software. Due to these features we use thrust API to implement different data structures.

5. The PGI suite is a commercial C/C++ compiler which provides several automatic and semi-automatic parallelization features. Further it incorporates a full CUDA CC++ compiler for targeting X64 CPUs. Even more important, it introduces the unified binary technology (PUB), which consists on the creation of a multi-target binary (GPU, INTEL CPU and AMD CPU) from an initial native CUDA code.

Cunetsim is a distributed GPU-based framework designed for wireless mobile network simulation. It aims to achieve two main goals: enabling extra large-scale network simulation and providing a significant speedup compared to traditional CPU-based frameworks. It considers that the GPU

architecture is adequate to hold the totality of a network simulation based on a CPU-GPU co-simulation [3]. It dedicates independent execution environments for each simulated node and uses the message passing approach through buffer exchange. The distributed design of Cunetsim is a fundamental cornerstone in support of heterogeneous computing architecture. In particular, the master-worker model was extended to a hierarchical architecture denoted as the coordinator-master-worker (CMW) model [4]. The default event scheduling policy handles the two types of event grouping described in section 3 but uses a centralized process and one data structure. Finally, synchronization and communication processes handle domain-specific operations between CM (Coordinator-Master) and MW (Master-Worker). As for the scheduling challenge, Cunetsim designers observe that its central event scheduler becomes the simulator bottleneck when the input event rate increases significantly. Moreover, they observe that a central event scheduler cannot handle several powerful GPUs. Therefore, a new scheduling approach becomes crucial to deal with those emerging architectures. Moreover, Cunetsim targets either CPU or GPU but not both at the same time; thus the H-scheduler concept presents a real asset to new frameworks generation.

In the remainder, we study the efficiency of the H-scheduler under extreme load and scalability conditions; therefore we use a very large scale network scenario which generates billions of events. First, we describe the evaluation scenario and setup and second we compare the efficiency of different scheduling policies under fair conditions. Finally, we analyze the performance of the H-scheduler where we detail the impact of each algorithm. In particular, we analyze the variation of the output event rate during the simulation, the decision path length and the variation of lookahead interval length during the simulation.

4.1 Scenario & Setup

We propose a large scale network experimentation scenario where we customize the benchmark methodology proposed in [3] by defining a static network topology composed of three independent activity areas (AA) each of which follows a grid configuration where the edge of an AA contains 750 nodes as illustrated in Figure 4; thus each AA includes 562.5K nodes[1]. The scenario includes one traffic source which generates 600 uniform 128-byte packets with 1 second of inter-departure time. All nodes forward unseen packets after a one-second delay to model the network latency whilst medium's reliability is reflected using dropping probability. Depending on the later, each node decides whether or not to relay a received packet. The drop probability (DP) is the parameter which allows us to introduce a random factor on the network behavior. To provide a valuable event rate while keeping a significant variation on the number of exchanged messages we use a DP of 0.1. The simulation duration is 5602 which ensures that the last generated message can reach the destination. In addition, we introduce a second scenario where we define for each node a randomly variable inter-departure-time in [0.1, 2] and a DP in [0, 0.28]. This scenario is used to study the robustness of the H-scheduler when the timestamp of different events present a significant entropy.

Even if these scenarios are outlying real networks and include major simplifications, we claim that they provide an important event number with a large rate variation as shown in Figure8. Moreover, they rely on a pool of simple events which did not require powerful computing resources individually, and use a simple implementation of both nodes and channels which provides a fair comparison later. Therefore, the main difference between considered approaches remains the scheduler efficiency. On the other hand, we use the Cunetsim framework, except for the NS3 case; thus all events are dual compliant with both CPU and GPU targets when we aim to use GPU. In what concerns the experimental context, the used frameworks are CUDA 5.0 and Open-MPI 1.4.1. The OS is Ubuntu Linux 11.10, the PGI compiler version is the 12.9 and the Nvidia driver version is 295.41. The hardware platform is one PC including an INTEL i7 3930k CPU (6 cores with hyper threading), 32 GB of DDR3 and three GeForce GTX860 2GB (1536 cores for GPGPU computing.

[1]That value represents the hardware limitation of the used GPU in term of memory space since each node needs 3.8 Ko

4.2 Comparative Evaluation

This section aims to highlight the efficiency of several scheduling approaches which differ by their execution targets and parallelism considerations. We distinguish three groups according to their execution targets respectively the CPU, the GPU and both, summarized in table 1. We use Cunetsim framework where we change solely the scheduler except for the NS3 simulator where we use the default scheduler. Moreover, we use the same implementation of nodes and channels; thus we guarantee a fair comparison. Based on the default scenario, we consider three metrics: the speedup with respect to a reference sequential execution, the hardware usage rate and the scheduling cost.

Table 1:

id	Target	Parallelism	Example
1	CPU	non	sequential
2	CPU	Op	NS3 scheduler [19]
3	CPU	Msv + Op	Cunetsim-CPU
4	GPU	Op	[17]
5	GPU	Msv	Cunetsim
6	GPU	Msv + distributed	D-Cunetsim
7	GPU+CPU	Msv + Op	H-scheduler

Table 2: List of different scheduling approaches, clustered their execution targets. Op =opportunistic and/or optimistic, Msv= Massive

Figure 5 shows the normalized speedup of each approach with respect to the sequential running time, computed using the average value of both sequential NS3 and Cunetsim. We observe that GPU-based approaches are extremely faster than all CPU-based ones. However we notify that the opportunistic approach on GPU (case 4) which presents an intermediate speedup (40x), did not consider the SIMD architecture of GPUs. In fact, its results are due to the efficiency of the hierarchical GPU memory and the existence of 24 independent SMX[2] in the used platform. On the other side, the GPU-based approach which relies on massive parallelism concepts(cases 5-6), presents an outstanding speedup which varies between 400x and 900x. Nevertheless, we observe that D-Cunetsim cannot reach the maximal expected

[2]The Kepler notation of the Streaming multi-processor

speedup (3*400x) while it uses 3 GPUs. Therefore we suspect that the scheduler is the main bottleneck since it cannot supply all GPUs with the correct rate. On the other hand, we observe that the H-scheduler is twice faster than the default scheduler while both use the same hardware. This considerable gain demonstrates that the hybrid scheduling approach reduces significantly the bottleneck impact.

Figure 6 presents the average hardware usage rate of each approach. It reflects the used resources to ensure the simulation. As expected, CPU-based schedulers ignore the GPU, however their CPU usage differs from one to the other. The sequential scheduler uses in average 20% of the CPU which represents the use of one CPU core. NS3 uses in average 84% of the CPU which means that the scheduler is not able to supply all available cores correctly while the third case (Cunetsim-CPU) ensures a full usage of the CPU. Therefore, we can assert that the event grouping policy presents a significant *added value*. The fourth case presents a mitigated score, it uses a small fraction of CPU and GPU resources but outperforms all CPU-based schedulers. This behavior confirms the previous observation.

In what concerns the dedicated GPU schedulers (5-7), we observe that the non distributed version of Cunetsim (case 5) uses efficiently one GPU and one core of the CPU; thus it does what it promises. We observe that the H-scheduler reaches the maximal CPU and GPU usage rate. In particular, it overcomes the default distributed scheduler which did not exceed 84% of the GPU usage rate while the H-scheduler approximates the 100%. These observations prove that the H-scheduler is able to maximize the usage of powerful solutions.

Figure 7 presents the average CPU usage rate of the scheduling process regardless of the simulation. Unsurprisingly, the H-scheduler needs in average between 6x and 8x more resources than CPU-based approaches and up to 3x more than GPU ones. This is due to the fact that it uses at least 4 different threads to achieve the scheduling process, in our case we use 6 threads distributed as follows: dispatcher, injector, one CPUscheduler and three GPUschedulers. Moreover, these threads work in parallel since the different data structures ensure the role of intermediate buffers. Therefore, the H-scheduler uses in average 30% of the CPU resources to ensure the maximal event flow.

We conclude that the H-scheduler is able to maximize the simulation efficiency, compared to a classical one; moreover it is able to deal natively with heterogeneous platforms if events are compliant. However, that efficiency requires additional dedicated resources compared to centralized approaches.

4.3 Performance Analysis

The H-scheduler is composed of several processes, and each process has two algorithms with the ability to switch between them autonomously, in addition, each algorithm reconfigures itself periodically based on a learning mechanism. In this section we propose to analyze the impact of each algorithm on the global behavior of the H-scheduler.

To analyze the impact of each algorithm we conduct the following experimentation series: First we measure the average event rate per simulated second, generated during the simulation across 100 runs. We realize so many runs because the shape of the curve of Figure 8 appears so perfect for a simulation including a random factor. Nevertheless we verify that the message propagation on the proposed network respects that shape. In particular, each peak of that curve reflects a maximal network activity in one AA. The second scenario which includes randomly variable connections between nodes, has a less regular curve[3]. Using the default scenario we process as follows: first we activate uniquely the advanced algorithm, second we activate the rapid one, third we activate the hybrid algorithm without any reconfiguration process or setting a timer and finally we consider the H-scheduler as described in section 3. The simulated time is 5602 seconds while the execution time varies between 413 and 670 seconds which complicates the comparison. To present an understandable representation, we normalize the output event rate with respect to the following formula $Outputrate = Nevents/Executionduration$. Corresponding results are shown in Figure 9. First of all, we observe that the advanced algorithm (red curve) is in general more efficient than the rapid one (green curve). However, the latter achieves punctually higher output rate, especially in the end of the simulation, characterized by a reduced number of messages. On the other hand, we observe that the hybrid algorithm provides better results while it introduces a large variability during the time as shown by the width of the curve (blue). We can assert that, when allowing each component to use the most adequate algorithm as a function of the situation, we reach a higher event rate with a risk of inefficient oscillation. Finally, the full H-scheduler which involves both hybrid algorithm and the continuous recalibration presents unquestionably the highest output rate but also the most variable behavior. In fact, we can distinguish four distinct and quasi-parallel sub-curves, each of which presents the maximal achievable rate of one of the available computing processors(1 CPU and 3 GPUs). We observe that the recalibration procedure allows the scheduler to match rapidly the typical hardware parameters which gives a significant gain of almost 30% compared to the hybrid algorithm only and about 90% compared to the default scheduler.

To illustrate the impact of the continuous re-calibration we propose to analyze the evolution of two parameters during the simulation: first we consider the average decision path length, computed as the average number of steps required to make the decision on where the event will be directed. Second, we consider the average scheduling interval length during one simulated second. As for the path length illustrated in Figure 10, we identify a first phase, where the length seems extremely variable, oscillating between 1 step (18%) and 5 steps (13 %). This presents a learning phase, which was arbitrary fixed to 500 simulated seconds. Afterwards, we observe a transition phase where the average decision path length decreases rapidly until reaching a steady state where the scheduling decision needs in average between one and two steps. We conclude that the continuous recalibration allows a significant gain in term of scheduling cost without compromising the decision quality. Regarding the scheduling interval length illustrated in Figure 11, we observe that the interval length is sensitive to the experimentation conditions; therefore its value decreases to 1-5 ms when messages number is high and increases to 10-15 ms when events are mixed. Considering that message events are na-

[3]We note that we vary the DP for that scenario between 0 and 0.28 because we loose the network activity beyond that threshold.

tively dependent, that behavior reflects in a faithful manner, the events relationship. Moreover, we observe that there is no learning phase since the unique rule during the whole simulation is to maximize event parallelism. Thus, an interval maximization phenomenon is observed in the beginning and the end of the simulation. This further confirms the bijective relation between the interval length and event dependency. Accordingly, we can conclude that the H-scheduler presents an interesting ability to deal with variable simulation rates under large scale conditions while maximizing the hardware usage rate even in heterogeneous context. Moreover, the re-calibration procedure and the dynamic behavior allow a significant support of the hardware characteristics without prerequisite knowledge.

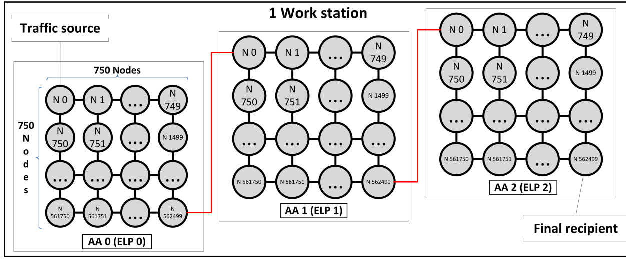

Figure 4: Topology of the benchmarking scenario

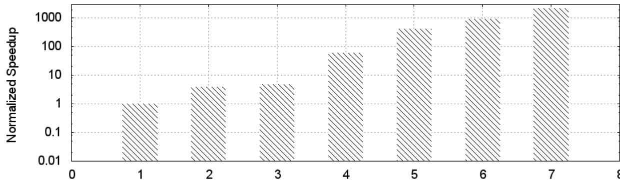

Figure 5: Normalized speedup with respect to the sequential runtime

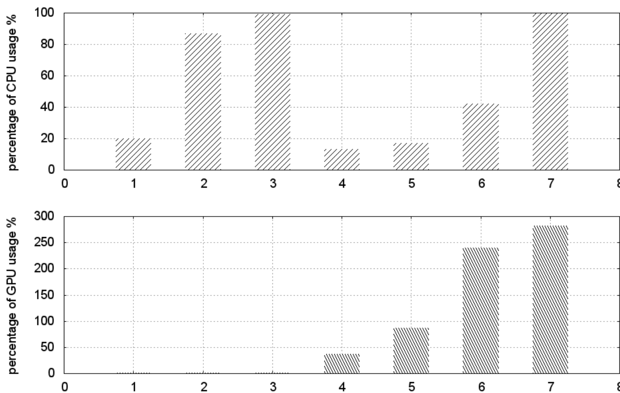

Figure 6: The hardware usage rate.

5. DISCUSSION

This paper presents a proof of concept where we aim to emphasis the potential of the H-scheduler. Thereby, we use

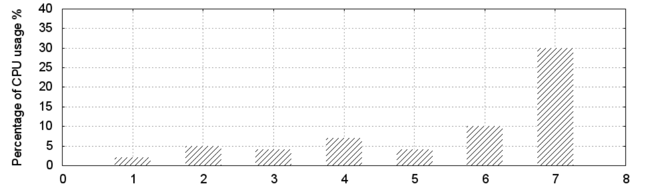

Figure 7: The scheduling cost.

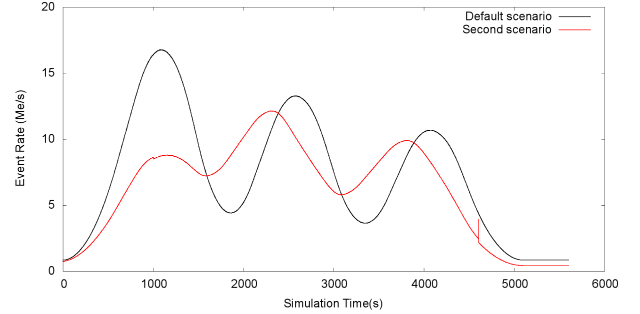

Figure 8: Variation of the input rate vs Time

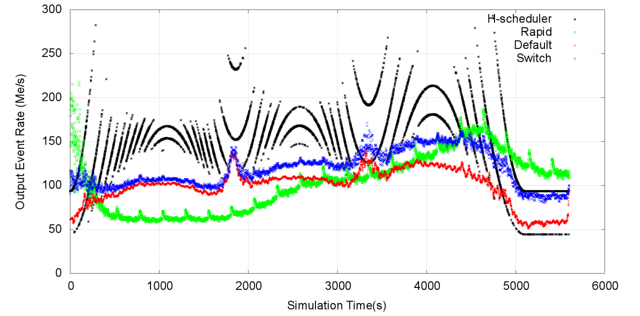

Figure 9: Output event rate of different algorithms.

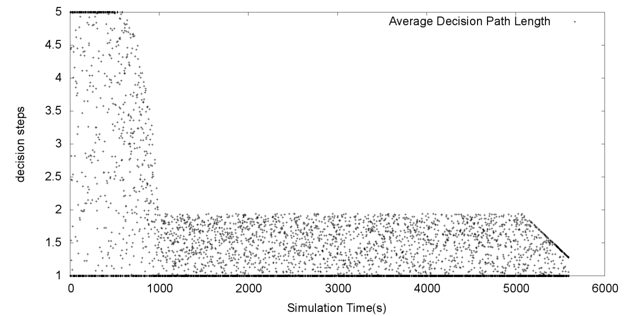

Figure 10: Average decision path length during the simulation.

an adequate machine including identical GPUs and well-designed scenario, suitable for high event rate. Nevertheless, we had identified five issues which need to be discussed: First, we did not consider the data structure filling issue. In fact, all events have the same importance whatever the status of the concerned buffer. An adaptive policy may use a RED [10] approach to manage different buffers. Second, the

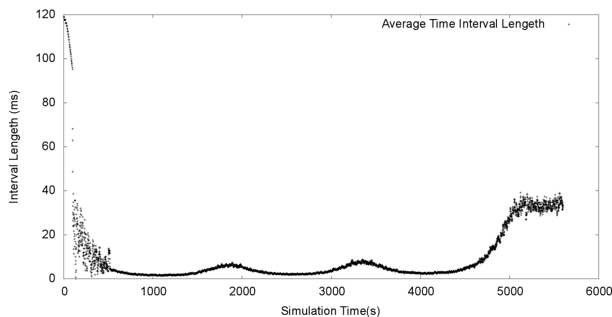

Figure 11: Average interval length during the simulation. It reflects closely the events dependency.

management of the algorithms switching relies on a loop-back mechanism which measures the load of each auxiliary resource. While this approach seems efficient to cope with very large scale simulations, analytical evaluation demonstrates that it may generate an instable system when the simulation pattern is unpredictable (such as the second scenario). Therefore, a weighted flow management appears as a reasonable stability compromise.

Third, we used in this work a limited range of computing hardware, including one GPU family and one Hexa-core CPU. We expect to evaluate the robustness of the H-scheduler under large heterogeneous conditions where we combine powerful CPUs (Xeon E5 2650) with professional GPUs and accelerators (Xeon Phi).

Fourth, we had observed that the brutal swing between both scheduling algorithm may be a source of instability; thus we had introduced a swing timer to stabilize the system. We expect that a smoothed transition between them may increases further the efficiency while providing enhanced stability. To conclude this discussion, we highlight that we rely on a data-abstraction mechanism which allows any event to access any data whatever its location. Therefore, the H-scheduler omits the data-access cost when computing the most suitable execution target. Nevertheless, we observe that the data locality is an emerging issue which may be considered on the event scheduling level, especially when targeting heterogeneous computer.

6. CONCLUSION

Discrete event simulation is widely recognized as an essential tool to analyze complex systems. Often, modern structures require sophisticated models while simulating a large number of entities in continuous interaction. However, the scalability of that simulation remains challenging, whatsoever in term of running time as well as in term of simulated entities. In that context, parallel (and distributed) discrete event simulation is actually the most relevant solution which allows a respectable scalability degree. However, event scheduling during the simulation over PDES requires optimized design to deal with emerging hardware innovation while respecting the simulation correctness. In particular, new computers are transformed into a heap of heterogeneous processors, each of which is more adequate for a specific task. Nonetheless, most of existing schedulers are derived from sequential concept which reduces their ability to cope with parallel hardware specifications. accordingly, the events flow is generally under expectation and the idle time is consequent.

We present in this work a new scheduling approach which aims to maximize the event throughput over heterogeneous computer while taking advantage of each processor specificity. Thereby, we rethought the event-driven simulation architecture to consider event flows rather than individual events. Therefore events are clustered as a function of their process and timestamps for the scheduling step while event flows are further directed to the adequate execution target. The implementation of our concept is denoted the H-scheduler referencing the heterogeneous computing; it considers particularly both GPU & multi-core CPU. In the current version, the H-scheduler is composed of four components: the dispatcher which computes the possible parallelization issues for events, the injector which determines the execution target of each parallel event group, the CPU-scheduler which ensures the execution of events on the corresponding CPU and the GPU-scheduler which ensures the execution of events on the corresponding GPU. Both injector and dispatcher present an abstraction layer which simplifies the user job. On the other side, both CPU-scheduler and GPU-scheduler are achieved according to a hardware/software co-design method; therefore we combine the simplicity of usage and efficiency of specific target-oriented solutions. Accordingly, experimental results demonstrate that the H-scheduler overcomes the majority of existing schedulers. In particular, it is able to achieve a reference simulation 1200x faster than the sequential one and 2x faster than the original Cunetsim scheduler while using the same workstation.

7. REFERENCES

[1] B. G. Aaby, K. S. Perumalla, and S. K. Seal. Efficient simulation of agent-based models on multi-gpu and multi-core clusters. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 29. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[2] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems*, pages 359–371, 2011.

[3] B. Bilel, N. Navid, and M. Bouksiaa. Hybrid cpu-gpu distributed framework for large scale mobile networks simulation. In *Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on*, pages 44–53. IEEE, 2012.

[4] B. Bilel, N. Navid, and B. C. Coordinator-master-worker model for efficient large scale network simulation. In *6th International ICST Conference on Simulation Tools and Techniques*, 2013.

[5] T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[6] L.-l. Chen, Y.-s. Lu, Y.-p. Yao, S.-l. Peng, et al. A well-balanced time warp system on multi-core environments. In *Principles of Advanced and Distributed Simulation (PADS), 2011 IEEE Workshop on*, pages 1–9. IEEE, 2011.

[7] R. Curry, C. Kiddle, R. Simmonds, and B. Unger. Sequential performance of asynchronous conservative pdes algorithms. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 217–226. IEEE Computer Society, 2005.

[8] G. D'Angelo and M. Bracuto. Distributed simulation of large-scale and detailed models. *International Journal of Simulation and Process Modelling*, 5(2):120–131, 2009.

[9] K. Dragicevic and D. Bauer. A survey of concurrent priority queue algorithms. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6. IEEE, 2008.

[10] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, 1993.

[11] R. Fujimoto. Lookahead in parallel discrete event simulation. Technical report, DTIC Document, 1988.

[12] R. Fujimoto, K. Perumalla, A. Park, H. Wu, M. Ammar, and G. Riley. Large-scale network simulation: how big? how fast? In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, pages 116–123. IEEE, 2003.

[13] M. Hybinette and R. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(4):378–407, 2001.

[14] J. Liu and R. Rong. Hierarchical composite synchronization. In *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pages 3–12. IEEE, 2012.

[15] H. Lv, Y. Cheng, L. Bai, M. Chen, D. Fan, and N. Sun. P-gas: Parallelizing a cycle-accurate event-driven many-core processor simulator using parallel discrete event simulation. In *Principles of Advanced and Distributed Simulation (PADS), 2010 IEEE Workshop on*, pages 1–8. IEEE, 2010.

[16] H. Park and P. Fishwick. A gpu-based application framework supporting fast discrete-event simulation. *Simulation*, 86(10):613–628, 2010.

[17] H. Park and P. Fishwick. An analysis of queuing network simulation using gpu-based hardware acceleration. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 21(3):18, 2011.

[18] J. Parker and J. Epstein. A distributed platform for global-scale agent-based models of disease transmission. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 22(1):2, 2011.

[19] J. Pelkey and G. Riley. Distributed simulation with mpi in ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 410–414. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.

[20] K. Perumalla. Parallel and distributed simulation: traditional techniques and recent advances. In *Proceedings of the 38th conference on Winter simulation*, pages 84–95. Winter Simulation Conference, 2006.

[21] K. S. Perumalla. Switching to high gear: Opportunities for grand-scale real-time parallel simulations. In *Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 3–10. IEEE Computer Society, 2009.

[22] P. Peschlow, M. Geuer, and P. Martini. Logical process based sequential simulation cloning. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 237–244. IEEE, 2008.

[23] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(2):157–209, 1997.

[24] W. Tang, R. Goh, and I. Thng. Ladder queue: An o (1) priority queue structure for large-scale discrete event simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(3):175–204, 2005.

[25] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[26] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pages 211–220. IEEE, 2012.

[27] S. Wang, C. Lin, Y. Tzeng, W. Huang, and T. Ho. Exploiting event-level parallelism for parallel network simulation on multicore systems. *Parallel and Distributed Systems, IEEE Transactions on*, 23(4):659–667, 2012.

[28] M. Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.