General Purpose Coordinator-Master-Worker Model for Efficient Large Scale Simulation Over Heterogeneous Infrastructure

Bilel Ben Romdhanne and Navid Nikaein
Communication System Department, Eurecom
06410 Biot Sophia-Antipolis, France
Email: firstname.name@eurecom.fr

***Abstract*—In this work, we propose a general-purpose coordinator-master-worker (GP-CMW) model to enable efficient and scalable simulation. The model supports distributed, and parallel simulation over a heterogeneous computing node architecture with both multi-core CPUs and GPUs. The model aims at maximizing the hardware activity rate while reducing the overall management overhead. The proposed model includes five components: coordinator, priority abstraction layer (PAL), master, hardware abstraction layer (HAL), and worker. The proposed model is mainly optimized for large-scale simulation that relies on massive parallelizable events.**

Extensive set of experiment results show that GP-CMW provides a significant gain from medium to intensive simulation load by exploiting heterogeneous computing resources including CPU and GPU. Regarding simulation runtime, the proposed GP-CMW model delivers a speedup of 3.6 times faster than the CMW model.

***Index Terms*—Heterogeneous Computing; Large scale simulation; Master-worker model; GPGPU; CUDA; PADS.**

I. Introduction

Stochastic simulation is used to study a wide range of applications from medical systems to the wireless communication networks. The simulation of such systems may have various objectives ranging from analyzing the system behavior to the validation of new concepts. Thus, the simulation becomes an essential tool on the development cycle of modern technologies. Even if simulation provides reproducible results, the scalability and applicability remain key challenges. In fact, increasing the size and more generally the realism level of the simulated system leads to a nonlinear increase in the required resources and the execution time, which in turn reduces significantly the simulation efficiency (Righter & Walrand 1989).

To speedup a large scale simulation, there are two main approaches: (1) parallelism and/or distribution of the simulation over several instances (also known as a logical process LP), and (2) usage of a dedicated accelerator to handle the bottleneck. The distribution of a simulation over multiple computing instances delivers a significant scalability gain at the cost of higher complexity and overhead. In particular, the ratio of overhead to guarantee the simulation correctness remains significant. In the literature, flat software architecture is initially used to define a parallel and distributed event simulation (PDES), where multiple LPs collaborate to perform the simulation. Such a design is widely used for small to medium scales (in term of number of LPs). However, its relative overhead depends on how applications are mapped to PDES and how the lookahead and the synchronization mechanisms are utilized. The overhead could rapidly increase as the simulation scale in terms of number of nodes, mobility rate, and traffic load becomes large.

To reduce such residual overhead when targeting a large-scale simulation, we introduce a two-levels hierarchical architecture, where a dedicated process (also known as the server) ensures the management of the simulation. The involvement of that process varies from one implementation to another. The master-worker (MW) model is an example of two-level hierarchical architecture that handles efficiently meta-computing systems (Park & Fujimoto 2012). Such a design is optimized for recent hardware; however, specific considerations must be taken to (i) increase the computing, data, and communication localities, and (ii) exploit the capability of a heterogeneous computing node. To deal with the MW limitations while coping with computational challenges of heterogeneous computing node architecture, a hierarchical approach was proposed in (Aaby, Perumalla & Seal 2010). Authors propose a new concept based on the interaction between CPU-based and GPU-based component. In addition, a specific consideration for the data locality was introduced. Nevertheless, that approach does not address the GPU memory restriction and induces a constant synchronization delay (Chen, Huang & Zhang 2012).

In this paper, we propose a general-purpose coordinator-master-worker (GP-CMW) simulation model, as an extension to our previous work (Romdhanne & Nikaein 2013), to radically increase the simulation efficiency when the number of events becomes large. The novel contributions of this paper are:

- managing the communication across multiple simulation instances through priority abstraction layer (PAL) to increase simulation stability,
- exploiting computing, data, and communication locality through hardware abstraction layer (HAL) to maximize simulation efficiency.

The proposed model considers the meta-computing system composed of several interconnected heterogeneous computing nodes as a system of subsystems. The main rule is to maximize

the interactions inside a computing node while minimizing the communication outside. In GP-CMW, at the top level, the coordinator ensures the global time synchronization and the load balancing among the masters. The master locally manages the time synchronization and event scheduling among the workers and interacts with the coordinator and other masters through PAL. The workers are the executing threads performing tasks communicating with master through HAL. Each addressing space has a dedicated HAL, which performs event scheduling across heterogeneous computing resources. From the coordinator point of view, the master manages one simulation instance, which is why the master-worker subsystem is refereed as an extended logical process (ELP). Note that in GP-CMW, the pair CM manages simulation across different addressing space (distributed simulation) while the pair MW manages the simulation within the same addressing space (parallel simulation).

The remainder of the paper is organized as follows. In Section II, we summarize the related works. In Section III, we describe the design elements and architecture of the GP-CMW model as well as its features. In Section III, we describe the design elements and architecture of the GP-CMW model as well as its features. Section IV presents the benchmarking scenarios and validation results for the GP-CMW model in comparison with the basic CMW model and MW. Section V discusses the applicability of the GP-CMW model for different type of simulations. Finally, Section VI concludes the paper and provides a summary of the contribution of this work.

II. Related Work

The digital simulation was introduced since the second world war for the requirements of the Manhattan project. The necessity of such a rapid and efficient validation tool was closely related to the computer usage. Since then, computer hardware has been evolved considerably from CPU-RAM-HDD, to a group of heterogeneous computing resources collaborating to perform a task. In parallel to this, the complexity of digital simulation has been significantly increased to provide a higher realism level and more reliable results. This calls for an efficient simulation capable of exploiting all the available computing resources to maximize the hardware activity rate. In the literature, various optimization techniques have been proposed that can be classified into four classes(Perumalla 2006): architectural optimization, local optimization, bottlenecks acceleration and hybrid optimization.

The architectural optimization attempts to efficiently parallelize and distribute the simulation over a set of computing nodes. In the flat design, different LPs are considered to be equivalent, and they collaborate to perform the simulation in a distributed fashion (Liu 2009). In fact, while previous works proof the scalability of the flat design, the mainstream of such work relies on small-to-medium LPs. Moreover, the internal communication inside each LP seems much more important than that inter-LPs. The clock synchronization is also a major scalability issue that was addressed earlier with the Chandy-Misra-Bryant algorithm (Fujimoto & Nicol 1992). Such algorithm copes well with both flat and hierarchical organization. Nevertheless, the scalability remains an issue in the flat design when the number of LPs increase (Fujimoto, Perumalla, Park, Wu, Ammar & Riley 2003). In the literature several optimizations, such as the lookahead (Fujimoto 1988) and the opportunistic and combined synchronization (Perumalla 2006), are proposed to reduce the idle time induced by the synchronization process. The two-level hierarchical design provides a solution to the scalability issue by introducing a centralized management service (called the server or master) in charge of synchronization and job assignment processes. The well-known example is the master/worker model compatible with meta-computing systems (Park & Fujimoto 2009). The main challenge here is the communication overhead caused by the non-locality of the master with respect to the worker when the simulation becomes large (i.e. in the order of several millions of simulated components). Furthermore, the master remains the critical bottleneck in such a setup as it drives the entire simulation. The multi-tier design addresses the scalability for heterogeneous computing nodes by partitioning them into several non-overlapping subsystems with one dedicated master (Wenjie, Yiping & Feng 2012). The number of tiers depends on the setup and available resources, which could potentially cause large synchronization delay due to cascading masters. This concept is extended to support GPU (Aaby et al. 2010), where the synchronization and communication overhead is significantly reduced in term of the number of exchanged messages. However, the delay remains an open issue in multi-tier architecture. Furthermore, the state vector mechanism remains existing and introduces a significant delay since each master manages larger works than traditional LPs (Pennycook, Hammond, Jarvis & Mudalige 2011, Chen et al. 2012), thus the latency issue needs to be addressed.

The local optimization aims at improving the efficiency of each LP in its environment. We distinguish two main trends: local parallelism and engineering optimization.

In general local parallelism acts at the event/instruction level to maximize the usage of multi-core CPUs or GPUs. The parallel event scheduling over CPU presents a reasonable tradeoff between the backward compatibility and the efficiency since it uses all available cores to execute in parallel future events(Liu & Wainer 2012). However, this approach relays on a unique central events list and one scheduler, which remains the bottleneck when targeting larger CPUs (e.g. INTEL MIC with 80 cores) (Satish, Kim, Chhugani, Nguyen, Lee, Kim & Dubey 2010). A similar approach introduced by Park et all (Park & Fishwick 2011, Park & Fishwick 2010) proposes to use the GPU as a multi-core computing co-processor. It relays on one central events list (CEL) and 8 threads, each of which runs on one core of the GPU and pops events from the CEL independently. However, that approach has two major limitations: first, it uses a central event queue which becomes inevitably the system bottleneck with larger GPUs, second, it considers a GPU core as a CPU one while the GPU is essentially based on SIMD architecture, where all threads must achieve the same routine. To handle this restriction, a dedicated GPU scheduling approach was proposed in (Romdhanne & Nikaein 2012), where authors use the event clustering approach to maximize the GPU usage while simplifying the

scheduling work. Nevertheless, this approach supports only one GPU that limits its scalability by that of the GPU in use. On the other hand, engineering optimization aims to maximize the usage of new hardware capabilities such as the different memory levels on the CPU and vectorial units. It acts mainly at the process/instruction level (i.e. the usage of the AVX instructions that allow the processing of 8 words per clock cycle maximize the performance of the re-wrote routines). A smart usage of that capabilities allows a significant performance gain (April, Glover, Kelly & Laguna 2003). Nonetheless, that approach is closely related to the implementation of each solution on one side and to the considered hardware on the other side.

Bottlenecks acceleration improves the simulation efficiency by offloading the computationally intensive tasks (typically identified through profiling) to a dedicated hardware such as DSPs, FPGAs or GPUs. The DSP is mainly used for signal processing and presents a real gain when the simulation considers physical phenomena such as radio signal simulation but does not offer a rich programming model suitable to perform the entire simulation. The FPGA provides a reasonable tradeoff between the efficiency and programming flexibility, and thus it is largely used to accelerate existing solutions. For example, Steenkister et all (Borries, Judd, Stancil & Steenkiste 2009) used the FPGA as a signal accelerator for wireless network simulation. The OpenAirInterface (Romdhanne, Nikaein, Knopp & Bonnet 2011) wireless technology platform provides an SDR implementation of LTE/LTE-A system using a full GPP model and uses FPGA to interact with the RF subsystem. The GPU offers a carrier solution that combines programmability and large computing power but requires a specific software architecture as its programming model is not fully x86 compliant. Despite this limitation, Perumalla et all (Perumalla 2009) demonstrate the feasibility of using the GPU as a simulation context and several works proof its efficiency as signal processing accelerator (Abdelrazek, Kaschub, Blankenhorn & Necker 2009, Bai & Nicol 2010). Other efforts have been given to provide an efficient processing solution based system-on-chip (SoC) and network-on-chip (NOC) or even a larger computing solution proposed recently by INTEL (Cramer, Schmidl, Klemm & Mey 2012). In particular, the XEON Phi co-processor (Jeffers & Reinders 2013) provides up to 64 CPU computing core per device. Such solution seems promising, and several recent works assert that its development-cost/gain tradeoff is interesting (Saule, Kaya & Catalyurek 2013, Heinecke, Vaidyanathan, Smelyanskiy, Kobotov, Dubtsov, Henry, Shet, Chrysos & Dubey 2013).

Hybrid optimization combines the benefits of the above-mentioned approached to achieve the simulation efficiency. This could be achieved through a new software architecture with massively local optimization and optimized libraries exploiting the heterogeneous computing resources. In this perspective, NS-3 is a well-known network simulator that combines new software architecture with massively optimized code (Lacage 2010). Further, new frameworks that rely on virtualized resources combine both architectural and local optimization to perform optimal usage of virtual and real resources (Yoginath, Perumalla & Henz 2012). Nevertheless, GPU and hardware acceleration solutions, in general, remain relatively new and weakly considered in such approaches. The CMW simulation model (Romdhanne & Nikaein 2013) follows the hybrid approach differently since it combines modified software architecture with the usage of heterogeneous computing resources including multi-cores CPUs and GPUs.

III. The General purpose Coordinator-Master-Worker Model: Architecture and Features

The GP-CMW model is a software architecture that considers a distributed large-scale simulation across heterogeneous meta-computing resources. It is designed around five components, as described below:

- Coordinator (C) is a top-level simulation CPU process with two essential tasks: load balancing and synchronization among all the active masters. In addition, coordinator provides user interfaces and simulation data collection services.
- Priority abstraction layer (PAL) manages the communication between the coordinator and masters as well as between masters. It separates the simulation control plane from the data plane.
- Master (M) is a CPU process and represents an intermediate entity of the simulation. It manages workers operating potentially on different computing resources within the same shared memory context and communicates with the coordinator and others masters through the PAL.
- Hardware abstraction layer (HAL) is a unique process per addressing space that manages the event scheduling through workers while the master generates events. In addition, the HAL configures the communication between different processes within the same addressing space to maximize the usage of the internal communication bus.
- Worker (W) is the elementary actor of the GP-CMW that performs the simulation routines and interacts with the input and output data. Typically, each worker is modeled with a finite states machine (FSM), has its own working data and handle incoming events according to its process.

In the GP-CMW, there is only one coordinator operating on N masters, each of which manages K workers. We denote each master and its associated K workers as an extended logical process (ELP). In the GP-CMW model, the simulation is first distributed over a certain number of workers (simulated components in common terminology) for the considered simulation scenario. Then, workers are partitioned into separate simulation instance according to the user-defined spatial and/or operating policies. Each simulation instance is managed by one master, and all workers interact with the outside world uniquely through the master, i.e. each simulated instance is performed by one master on one computing nodes. To maintain the simulation stability, the PAL gives the highest priority to the control plan communication, in particular, synchronization messages. It relies on MPI primitives. The HAL incorporates a dedicated event scheduler per execution target, namely different GPUs and multicore CPUS, and determines for each event the most suitable execution target depending on current

load and the type of event so as to maximize the hardware activity rate (Romdhanne, Bouksiaa, Nikaein & Bonnet 2013). Figure 1 illustrates the hierarchical architecture of the GP-CMW model.

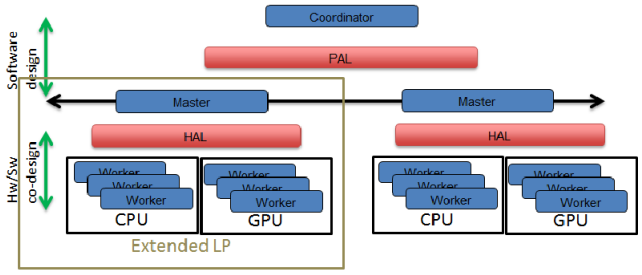


Fig. 1. GP-CMW simulation model

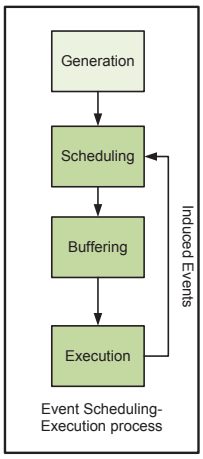
In the following section, we first detail the event management model, with particular attention to the concept of event-flow. Then, we present the synchronization mechanisms followed by hierarchical communication model.

A. *Events Management: Modeling, Scheduling, and Execution*

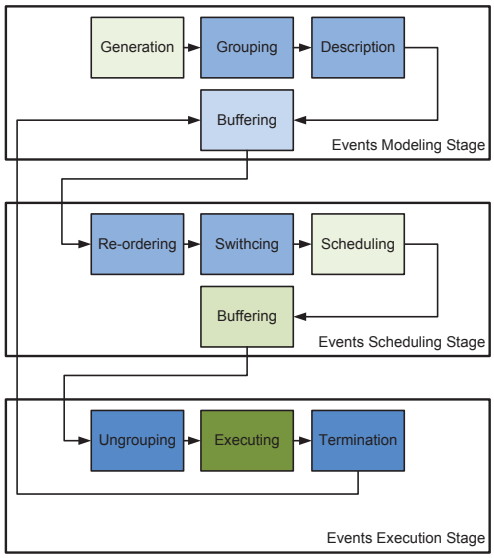
In discrete event simulation(DES), an event represents the execution of one state, activity, model or algorithm with particular parameters. A main activity of the simulation process consists of the management of events. A typical event life cycle in DES includes at least four steps: *generation*, *scheduling*, *buffering* and *execution* (see Figure 2(a)). By agreement, there are two types of generated events: those defined by the user scenario (i.e. periodic events) and that generated recursively due a previous event execution. In both cases, an event has specific timestamps that define when it must be executed refereeing to the simulation time. A sequential scheduling process consists of sorting incoming events based on their timestamps and buffering them into a FIFO list to be executed. A basic parallel scheduling process aims to execute events in parallel without violating the event casualty (conservative approach) or allowing violations within the lookahead horizon and recover from them (optimistic approach) (Fujimoto 1990, Som & Sargent 1998, Quaglia & Cortellessa 2000, Deelman, Bagrodia, Sakellariou & Adve 2001). Advanced parallel scheduling algorithms analyze events dependency to maximize the parallelism rate while conserving the simulation correctness (Wenjie et al. 2012). In the case of parallel scheduling for multiple execution targets, a central scheduler becomes an imminent bottleneck. In fact, such a situation can be modeled as a workflow with many consumers, and one producer [1: N], where consumers are the execution processors, and the producer is the scheduler. Therefore, increasing the number of consumers with one-to-one exchange order results inevitably in a famine situation at the consumers level and a bottleneck at the producer level. Moreover, one critical issue of parallel scheduling process remains its high cost that increases rapidly as a function of the number of events and the number of execution processors. Thereby, the majority of parallel schedulers deal with limited numbers of simulated elements (thousands per LP) and execution processors (typically 6-64 in the current state of the art).

In contrast to traditional approaches, the GP-CMW event management introduces three new features: (1) the separation between the event modeling, scheduling, and execution, (2) the events flow, and (3) the event grouping/ungrouping. In fact, events are modeled using an enhanced descriptor that includes event dependency meta-data, in addition to common information, which in turn simplifies the parallel scheduling process. In the remainder, we denoted the creation of the event descriptor and its enhancement with additional data as the event modeling. Indeed, the events management process on the GP-CMW separates physically between the event modeling, the scheduling, and its execution in that the modeling is achieved by the master, the scheduling is performed by the HAL and the execution by the pool of workers. Furthermore, it relies on the massive parallelism concept which is a suitable software model for SIMD hardware, and in particular for GPGPU programming. The main idea consists of generating cloned events, each of which performs the same operation on independent data. The GP-CMW model combines the generation of cloned independents events (CIE) with the detection of foreign independent events (FIE) that differs in both operation and data if they can be executed in parallel to maximize the simulation efficiency. Finally, to bypass any potential bottleneck due to the increasing event rate, the GP-CMW model allows compression of CIE events into one entry. This simple yet efficient technique provides a significant gain in terms of scheduling cost and time complexity. To realize this approach, we propose to modify the event life cycle with three independents stages: the events modeling, the events scheduling, and the events execution (see Figure 2). These stages are interconnected through intermediate buffers that maintain a continuous events flow between the event producer and consumer in order to increase the system stability.

1) Events Modeling Stage: On the event modeling stage, the master preprocesses incoming and generated events in order to reduce the complexity of scheduling stage. Thereby, it includes four steps: the generation, the buffering, the grouping and the enhanced description. First, the master generates initial events according to the user-defined simulation scenario. In particular, the desired timestamp of each event is generated. In parallel, it processes incoming events during the simulation. These events can be recursively generated by other events under execution or received from outside. On the second step, the master groups CIE into one entry. The main issue of the grouping step is to be lossless and reversible. In particular, the inclusion of events parameters determines the efficiency of the procedure. Figure 3 presents two grouping cases, case A is non-optimized and generates additional work while the case B is optimized and simplifies both the grouping and ungrouping step. We assume that the user must provide identical parameters for CIE when describing the simulation. In the last step, the master extends the event descriptor with additional information, including (i) event dependency information, (ii) event execution timestamp, (iii) I/O data access,



(a) Standard event life cycle in the DES



(b) GP-CMW event life cycle

Fig. 2. Events Life Cycle

(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 (Romdhanne et al. 2013). 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. We note, however, that the HAL will compute this timestamp at the execution stage according to the GP-CMW design. This presents a main contrast with the basic CMW model and highlights a complete separation between the event description and execution. The I/O data access defines the permissions given to an event to read and/or write a shared memory area. Finally, the execution target defines where an event could be executed, CPU, GPU or both. Once creation steps concluded, the event descriptor will be buffered into a FIFO buffer (technically denoted as a future event list: FEL).

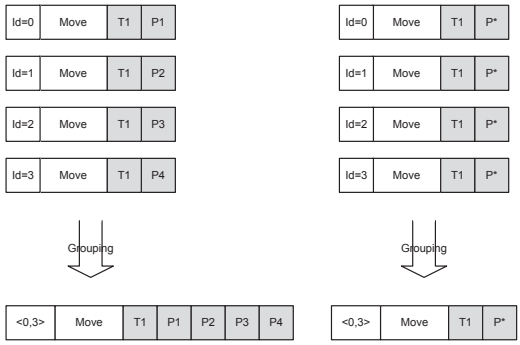


Fig. 3. This figure presents two grouping cases: (1) if the events to be grouped have different parameters, then the grouping process will concatenate them sequentially, and (2) if all events have the same parameter (e.g. share the same pointer to a common structure), then the grouping process generates one parameter for all

2) Events Scheduling Stage: On the event scheduling stage, the scheduler decides where and when each event will be executed. The decision is quite complex due to two reasons: first, execution targets may have different inherent characteristics including computing power, memory, and parallelism nature. Second, attributing a new execution timestamps to different events while maintaining the simulation correctness implies the usage of sophisticated algorithms. In contrast with the basic CMW model, the GP-CMW introduces a hardware abstraction layer (HLA) that ensures the scheduling task. It incorporates the H-scheduler that has been presented on (Romdhanne et al. 2013). The H-scheduler is composed of four main processes: the dispatcher, the injector, the GPU-scheduler, and the CPU-scheduler, where events are flowing(see Figure 4). In fact, the event flow concept is a main feature of the H-scheduler. It consists of detecting any eventual bottleneck in the fly and resolving it. Typically, each process has one (or more) input buffer and one or more output buffer. The divergence between their respective feeling rates indicates the existence of a bootblack in the concerned process. Thus, a rapid algorithm is activated to maximize the events flow through that process until reaching a stationary load.

The entry point of the scheduling stage is the dispatcher that pops events descriptors from the intermediate buffer(s) and pushes them to the correct place of a 3-dimensional buffering data structure called the 3D-AL. It includes a list of sub-lists, each of which represents a time interval that includes a group of parallelizable events. A sub-list is defined by a beginning time(bTime) and an ending time (eTime). An entry of a sub-list is either grouped events (CIE) or a unique one, according to its generation method. Entries of the same sub-list are assumed to be parallelizable even if they have different timestamps. Two events are dependent if that information is explicitly included into their descriptor or if they share data. The resolution of data dependency can be done using three methods: (1) splitting the sub-list, (2) merging dependents events into one sequential entry and (3) transforming the sub-list into a time ordered one. This step is denoted as the reordering step since each event will have a new execution timestamps. The next step is the switching, it consists of

choosing the most adequate execution target of each entry. Concretely the injector process pops events descriptors from the 3D-AL and injects them into the corresponding secondary array list denoted as 2DAL. Each execution target has its own sub-scheduler and 2DAL. The injector can distribute events of one sub-list through available resources. In such case, it uses a checkpoint system to synchronize the simulation. The switching decision relies on the parallelism level to choose the target class (CPU or GPU) and the hardware usage rate to choose an instance of the class (GPU1 or GPU2 for example). The 2DAL data structure is particularly optimized for the parallel execution over heterogeneous computing resources. It has to be mentioned that there is two different type of 2DAL, one for CPU targets and one for GPUs. To conclude this stage, each dedicated sub-scheduler manages its own 2DAL without any timing control according to a best-effort algorithm. It ensures event execution and presents in that sense a transversal process between the scheduling and the execution stage.

In simpler words, during the scheduling stage, the H-scheduler pops events from the FEL and switches them to one of the dedicated sub-schedulers. The dynamic strategy of the execution target choosing is a central feature of the H-scheduler. It relies on two parameters: the parallelism size of CIE entry and the saturation rate of computing resources. Thus, extremely large CIE are automatically switched to GPU while FIE are directed to CPU. The interpretation of what is extremely large relies on the GPU capabilities. On the other side, if the system includes different GPUs the switching step will rely on several thresholds, such that all computing resources will be feeds on the correct rate to avoid both famine and saturation states.

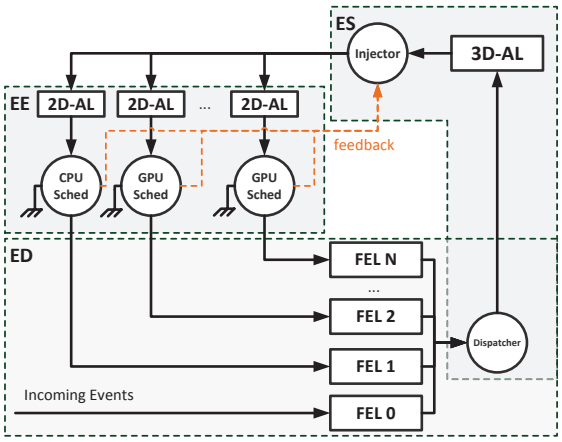


Fig. 4. Design elements of the H-scheduler, where the incoming events present what the event scheduler receives from outside, including what it receives from the master, and the feedback mechanism that drivs the injector decision.

3) Events Execution Stage: On the event execution stage, all sub-schedulers act simultaneously to increase the event processing rate. In what concerns the CPU-scheduler, it first creates as many threads as available cores. Then, it pops events from the corresponding 2DAL and assigns them to the first available thread. If the CPU-scheduler detects grouped events in its 2D-AL, it relies on the OpenMP API to ensure their parallel execution. If an executed event generates a new event, such event might be re-executed by the sub-scheduler in the same time window of the active time interval if the simulation correctness can be maintained. Otherwise, the newly generated events will be fed into FEL for future execution. It has to be mentioned that the basic CMW model relies only on the second approach and that a non-negligible speed up of up to 10% in term of runtime and 4% in term of hardware activity rate can be achieved.

The GPU-scheduler is slightly different since it relies on a hybrid software-hardware scheduling mechanism. At the software level, it handles always grouped entries that are translated to the CUDA calls with predefined generic parameters. The CUDA driver will then generate the corresponding threads and sending them to the GPU. At the hardware level, the embedded GPU GigaThread-scheduler first distributes events threads blocks to various SMs and second assigns each thread to an SP inside the corresponding SM.

B. The GP-CMW Synchronization Mechanism

When targeting distributed simulation, the system synchronization guarantees that there is an independent mechanism that ensures the time coherence of the simulation across different machines. Such a mechanism defines a reference time to be synchronization and a clock-advancement to drive the simulation. In general purpose simulation that uses discrete events concept, there are three distinct notions of time: (i) the physical time, representing the duration of the physical phenomena that we model or the time that the modeled real event requires to be performed (ii) the simulation time, representing the physical time in the simulation, and (iii) the wallclock or execution time, which is the elapsed real time during the execution of the simulation as measured by the hardware clock. Accordingly, the majority of the existing conservative mechanisms proposed to synchronize the simulation time. In what concerns the simulation time advancement, there are two methods relative to the DES: the time-driven and the event-driven. In the time-driven simulation, the clock increases sequentially from one value to the next with a predefined granularity which defines a kind of time interval. In distributed context, that model relies on the acknowledgment of each elapsed interval by each involved process. The acknowledgment can be explicit using NULL message or implicit using the communication timestamps. In the event-driven simulation, the clock jumps from the current event-timestamp to the next. Such a model avoids crossing empty intervals but implies a sophisticated synchronization mechanism in a distributed context. In general purpose distributed simulation the lookahead is the key ingredient for all conservative synchronization methods, which can be defined as the ability of a simulation instance (also known as a logical process in the literature) to predict future behavior when modifying the event lists of other instances (LPs). More precisely, an instance P has a lookahead with respect to an instance Q if the simulation clock of the instance P is at time s, and yet P can determine that under no circumstances an event will be inserted or deleted from the event list of the instance Q with timestamps $t > s$ (Nicol 1996).

In the GP-CMW model, we aim at maximizing the efficiency of the distributed simulation while minimizing the management overhead. Therefore, we adopt a hybrid and hierarchical synchronization mechanism where we separate the coordinator-master plan from the master-worker plan. Each component of the model has its own clock which progresses independently. The coordinator clock advances according to a time driven model. It defines a group of checkpoints that must be acknowledged by all active masters. The elapsed time between two consecutive checkpoints represents one work unit (WU) of the distributed simulation.

Regarding the coordinator-master plan, there are three main messages: S, E, and R. The message S is initiated by the coordinator and used to notify the masters to start a new WU. The duration of each WU is indicated in the message parameter. During the execution of a given WU, the master may request a specific duration for the next work-unit based on the synchronization algorithm of the master-workers plan. At the end of the current work-unit, each master sends an E message back to the coordinator to acknowledge the end of WU. The duration of each WU can be either static defined at the beginning of the simulation or dynamic calculated at each WU (indicated by the R message). When receiving an R message requesting a WU duration, the coordinator recomputes the duration of the next WUs according to maximize the global efficiency. Several user-defined policies can be applied to compute the WU duration, such as min, max, average, and the most requested.

This active and conservative process implies a significant overhead, but it simplifies the management and control of the distributed system. Nevertheless, an adaptive WU length may reduce the impact of such overhead (i.e. in network simulation benchmarking (Romdhanne, Nikaein & Bouksiaa 2012) using a small WU of 1 second induces a runtime increase of 10% compared with a WU of 10 seconds).

The master clock is also time driven and is incremented according to the time intervals computed by the events scheduler. Each master provides a list of CIE and FIE related to the current WU to HAL. Based on WU duration, HAL determines a set of independent and contiguous intervals used to (i) execute events belonging to the same interval timestamps even if their timestamps are different, and (ii) to advance the master clock (implicit synchronization). It has to be mentioned that during a WU, each master can progress according to its inherent speed, which in turn may not guarantee the simulation correctness. Therefore, we apply an optimized lookahead protocol to ensure the correctness among all ELPs (Shmueli & Feitelson 2003). Finally, the worker clock is the unique element which is event driven. When a worker is assigned to execute an event, it leaves the idle state to become active and then updates its clock with that of the master clock. Thus, the worker clock advances on per event-basis (only the clock of active workers are updated), and that the clock jumps from the current timestamps to the next one similar to traditional event-driven simulations.

C. GP-CMW Communication Model

In GP-CMW, the communication between different simulation entities is done through the message passing. It implements a hierarchical communication model allowing workers belonging to the same (local) and different (foreign) simulation instance(s) to interact among each other. The GP-CMW extends the basic CMW model and exploits locality through dedicated resources (e.g. bus, memory, ram) with the same addressing space. Four cases are possible as describes below:

1) If both workers are in the same GPU block, the message is written on the GPU shared memory and a reference is given to the destination. If there is more than one destination, the message is written to the in-buffer of the destinations ensuring that each worker has the entire control of its message.
2) If both workers are on the same GPU but on foreign blocks, the message is written on the GPU global memory, and a reference is given to the destination. Multiple destinations will receive distinct messages as in the first case.
3) If both workers are in the same ELP (i.e. the same memory space), the message is written on the destination in-buffer using direct (CUDA) communication. In this case, the destination can either be a CPU worker or a worker in a different GPU.
4) If both workers are in two different ELPs, the source worker writes the message on an intern out-buffer of the master. The source' master will transfer the message to the destination master.

It has to be mentioned that the HAL incorporates a routing table that includes the identity of workers and their physical location, and it used to select the appropriate communication method. Such information might be cached at the worker to reduce the latency before communication can be established.

1) Further Considerations: To further optimize the communication model, PAL implements three additional methods, namely packet aggregation, priority management, and multicast message passing. With the packet aggregation method, a master does not initiate a communication with other masters for every single request. Instead, it will start the communication when the waiting time window expires, or the number of buffered packets reaches the given threshold. While such method introduces an addition delay to accumulate multiple packets into a single packet before being transmitted, it pays off when a large amount of small packets are exchanged among several ELPs. Please note such an operation is transparent to a master in that PAL constructs an aggregated packet on per destination (i.e. master) basis and that it performs the disaggregation operation before a packet is received by a master.

The priority management method is used to separate simulation control plane from the data plane by associating and managing different priorities flows. In particular, synchronization, communication management, and load balancing control plane message flows have a higher priority than the date plane flows. Through this method, PAL ensures the timely delivery of critical flows under a heavy load and maintains the simulation stability (Lacage & Henderson 2006).

The last method, multicast message passing, handles packet duplication for each destination identified by the multicast address. Two approaches exist, namely smart pointer and real

independent copies, where the latter minimizes the memory usage with the risk of bottleneck creation and the former avoids central management at the cost of a memory usage (Wolf, Cai, Huang & Schwan 2002). The GP-CMW adopts the real independent copies to remain natively compliant with both CPU and GPU. As in the previous case, the packet duplication process is performed by PAL and remain transparent to masters.

IV. Performance Evaluation

Previous studies on the performance of CMW demonstrate that a significant gain is achievable compared to existing simulation models, namely flat and master-worker (Romdhanne & Nikaein 2013, Romdhanne et al. 2012). While the benchmarking scenario used in CMW is based on a grid network with a flooding protocol, in this work, we consider a complex scenarios with extremely high event generation rate that simulates a popular massively multi-players on-line game named as *Command and conquer, Tiberius alliance*.

We evaluate the performance of the proposed GP-CMW with the CMW (denoted as basic CMW) and the baseline master-worker model operating on multicore CPU (denoted as MW-CPU) using Cunetsim framework.To analyze the benefit of PAL and HAL, we present the results for two additional configurations, namely CMW-PAL and CMW-HAL. Note that GP-CMW incorporates CMW, HAL, and PAL within the same simulation model. For comparison, the following metrics are considered including simulation runtime, synchronization delay, communication latency, and hardware activity rate.

In the following subsections, we present the game model, simulation scenarios, and setup followed by the experimentation results for the considered metrics.

A. Game Model

The command and conquer game is defined by a group of independent worlds, each of which is represented by a $1000*1000$ grid. A world is initially occupied by the forgotten, which have their own infrastructure to manage resources. They have several types of military bases with increasing complexity level, from 1 to 50. The ultimate goal of the game is to control the world. When a player integrates a world he/she has one initial base with limited resources and must develop the base, army and defense over time. Players can create, integrate or leave an alliance. Members of a given alliance share the controlled resources and infrastructure. However, the alliance size is limited to 50 players creating a competition to control resources and infrastructures across the worlds. Moreover, alliances can define a diplomatic relationship with others. The development of players can rely on free available resources, though limited on time, or on paid resource packets [1]. The Financial stock of the game is to sell such packets during the life of a world, which may last up to one year. Predict the evolution of a world using analytical tools is a complex task as a large number of parameters are evolving over time based on the player and alliance interactions. For example, a war between two alliances may double the number of connections, the connection duration and also the number of sold packets.

When monitoring the evolution of the game, we notice that a significant fluctuation in gaming experience related to server availability, e.g. during a three-month observation, servers error and maintenance happened 20 times lasting 30 minutes, the website was offline two times for 3 hours. This calls for an accurate prediction of game dynamics to allow (quasi-)real-time servers and resources provisioning under a massive number of players with a high rate of interactions.

To simulate such a scenario, we extend the Cunetsim framework with new models to capture the game dynamics. Two agents are defined: (1) a player, which simulates the behavior of an individual user in terms of mobility model, communication pattern with other players, and number of connections with the backend servers, and (2) an alliance, which simulate the behaviors of a group of players as a whole in terms of diplomatic relationship pattern, and resource control. To increase the accuracy of the models, we observe the behavior of 1000 players and 30 alliances which evolve in French worlds number 2,6,7 and 10 and apply a trace-driven modeling methodology to fit the models.

B. Experimentation Scenarios and Setup

Two reference scenarios are designed ranging from medium-to-high (scenario 1) to high-to-intensive (scenario 2) simulation load. Both scenarios simulate 144 worlds during one year, where each timestamp represents 1 minute (i.e. 525600 total timestamps). The number of base per player is uniformly distributed between 1 to 20. Only 10% of players communicate with different worlds. For the first scenario, the number of players per Worlds is set to be uniformly distributed between 2k to 5k, while for the second scenario is between 25k-50k.

All the experimentation are carried out using the largest European GPU-based super-calculator, the TGCC Curie infrastructure. In our setup, each world is represented by one independent ELP and all simulation instances are synchronized at each timestamp, which in turn allows monitoring the impact of game dynamics in all worlds with the granularity of timestamps.

C. Results

1) Simulation Runtime: The simulation runtime is a measurement of the physical time needed to perform the totality of a given simulation and represents a global simulation efficiency. In the first scenario, the average number of total players is 350k, and the average number of the total alliance is 10k, which generates approximately 2 Tera events for the entire simulation time. In the second scenario, the total of 2M agents including players and alliances exist generating up to 50 Tera events during the simulation.

Figure 5 illustrates the simulation runtime of GP-CMW, basic CMW, CMW-PAL, CMW-HAL, and MW-CPU as described at the beginning of Section IV. When comparing the relative gain with respect to the basic CMW, we observe that CMW-PAL and CMW-HAL improve the simulation runtime by 12% and 7% for the first scenario, and 50% and 100% in

[1] www.ea.com/fr/sports

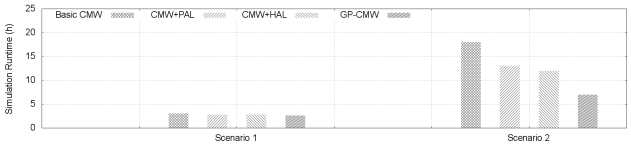


Fig. 5. Simulation runtime and the impact of the PAL and HAL.

the second scenario. When comparing the basic CMW with the GP-CMW, we notice when both PAL and HAL are enabled, an improvement of 19% for the first scenario and 260% for the second scenario are achieved. The results show that both PAL and HAL provide a significant gain in simulation runtime especially when the event rate becomes enormous. It can be seen from the figures that the gain obtained through HAL is higher (almost twice) in both scenarios. This is because HAL exploits both data and communication localities across all the available computing resources when scheduling events, which in turn maximizes the hardware activity rate. When comparing with GP-CMW, the gain of both layers are preserved indicating their complementarity.

2) Synchronization Delay: Synchronization delay is defined as the elapsed wallclock time between the reception of the first and the last acknowledgment of the current WU. It represents the maximum waiting time experienced by the masters and all its workers (entire ELP) and is measured at the coordinator level. Figure 6 shows the statistics of synchronization delay for the considered scenarios and configurations. We observe a relatively constant synchronization delay ranging from 0.26ms to 0.28ms with 0.19ms variance for medium-to-high simulation load, and more variability, ranging from 0.48 to 2mn with 1ms variance, for the high-to-intensive simulation load. This proves the effect of load on the synchronization delay, and consequently on the activity rate of the masters. In the high load scenarios, such a variability in synchronization delay is one of the main cause of simulation instability that has to be managed. In both scenarios when PAL is applied, such variability is not observed, and the synchronization delay remains constant, i.e. between 0.26ms and 0.28ms.

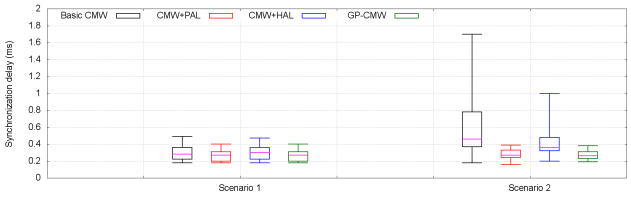


Fig. 6. Synchronization delay. We use the candle-sticks representation, where the box presents 95% of measured values during the simulation. The average value is the line in the middle of the box while the max and the min values give an overview of the overall variability during the simulation.

3) Communication latency: We define the communication latency as the elapsed time between the transmission of a message by the sender and its reception by the receiver. This is a real time value, relative to the simulation framework behavior and performance. In a distributed and parallel simulation, we distinguish between two communication categories: (i) between distinct ELPs known as inter-ELP communication and (ii) between simulated entities of the same ELP known as intra-ELP communication. We compute two values for each worker (player and alliance) and master (ELP). We rely on a statistical representation that summarizes the communication latency for both scenarios.

Figure 7 presents the measurement of inter-ELP communication for the considered configurations. We observe that the basic CMW requires on average 5 ms to exchange a message between two distinct ELPs, whereas when PAL is applied the delay becomes five times lower. This is due to the two additional methods supported by PAL, namely packet aggregation and multicast message passing, allowing to increase the performance among ELPs. As for HAL, the performance remains the same as basic CMW but with lower variability.

Figure 8 shows the measurement of intra-ELP communication for the considered configurations. We observe that the basic CMW model requires on average about 5 us to exchange message within an ELP, while with HAL, the performance is significantly improved, ie. up to 10 times faster. This is because HAL exploits the direct memory access and dedicated memories allowing workers to collaborate without CPU involvement.

From both figures, it can be seen that HAL and PAL are complementary allowing to achieve a significant gain in both parallel and distributed context.

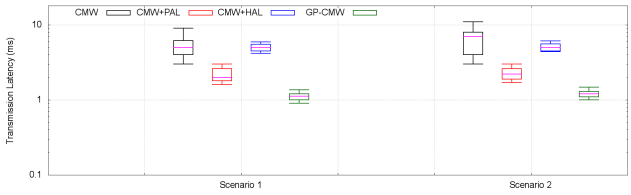


Fig. 7. Average Inter-ELP communication latency.

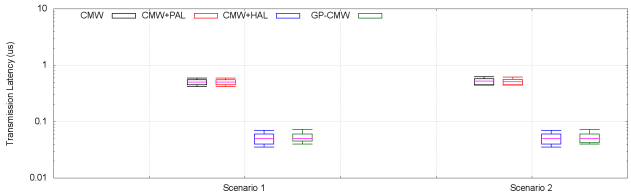


Fig. 8. Average intra-ELP communication latency.

4) Hardware Activity Rate: The hardware activity rate is a parameter that measures the activity rate of available resources during the software execution. In this study, we focus on four components: the CPU, the GPU, the RAM and the GDRAM. We rely on the OS primitives to evaluate the activity rate, and we compute the average value during the simulation.

Figure 9 and Figure 10 present the average CPU and CPU activity rate during the simulation. We observe that when HAL is enabled, both GPU and CPU activity rate also increase. This is mainly due to the efficiency of the hybrid scheduler that (i) exploits all the available cores to perform the simulation, and (b) schedules events by taking into account the event

type, CIE or FIE, as well as the instantaneous load of each target. By design, HAL schedules the FIE events, i.e. isolated and/or recursive events, for CPU target, and CIE events, newly generated event that only differ in data, for GPU target. Please note that the basic CMW does not distinguish between CIE and FIE, and thus schedules all events either to CPU or GPU.

Figure 11 presents the CPU memory load (RAM) during the simulation. It can be seen from the figure that the memory usage rate is load-dependent, and as expected the GP-CMW requires up to 50% more memory than the basic CMW under high-to-intensive simulation load. This is mainly due to the usage of intermediate buffers, used to maintain the events flow stability. Nonetheless, the memory load remains below 100% for scenario 2 indicating that inter-ELP communication may not be fully loaded. Please note that only 10% of the players communications with outside worlds. The GPU memory load (GDRAM) is shown in Figure 12. We observe the same trend as in case of RAM except that the memory usage is maximized for all configuration in scenario 2.

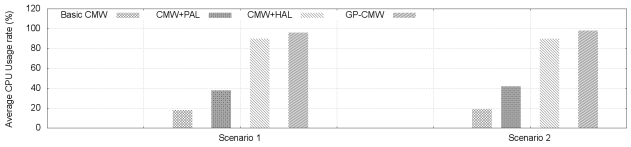


Fig. 9. Average CPU activity rate.

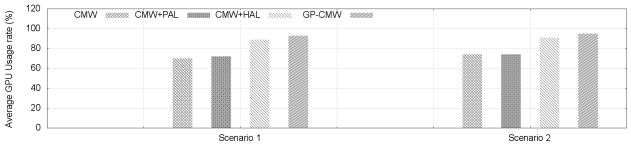


Fig. 10. Average GPU activity rate.

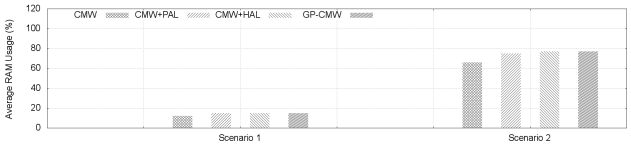


Fig. 11. Average RAM usage rate.

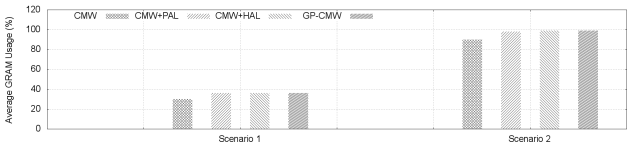


Fig. 12. Average GRAM usage rate.

V. Discussion

In the previous section, we highlight the advantage of using a GP-CMW model to simulate a large scale online game characterized by a massive inter-ELP communication and sporadic intra-ELP communication. In this section, we identify under which simulation scenarios the proposed GP-CMW model presents performance gain when compared to CMW and MW. For benchmarking, we reuse the Cunetsim framework as it supports not only GP-CMW and CMW but also MW for both CPU and GPU. The hardware platform consists of six standard workstations with the following configuration: Intel i7 6950X CPU core, 64 GB of ECC DDR4 RAM, 2 GPUs one K80 and one P100, and 1Gb and 10Gb Ethernet interfaces. We rely on the Intel core *i* family rather than XEON in order to adjust the CPU frequency on-the-fly (also known as dynamic frequency scaling or CPU throttling) from 1GHz to 4.5Ghz.

The benchmarking scenario is a large scale wireless mesh network with no node mobility in a flat area of 90km x 60 km, divided into six subzones of 30km x 30km each. There are 24k nodes randomly distributed in the space. Each subzone will be simulated in one LP or ELP depending on the considered architecture, and each one will simulate 4k nodes. Each simulated node implements one `http` server and one client; each `http` client is attempting to connect to a random server. Three parameters are considered to benchmark CMW and MW based simulation models, namely (i) the density of CIE, (ii) the density of recessive events, and (iii) the frequency of CPU, which are evaluated in the following subsections.

A. Density of CIE

In this experiment, we vary the density of CIE over all the simulated events. The density is defined as a percentage of the grouped events in the simulation. Note that the grouped events can be represented using one event descriptor, but their density remains the same. The hardware is configured with 3GHz CPU frequency and 10Gb network interface. The simulation is executed only one time to avoid recursive events.

Fig. 13 summarizes the simulation time for each simulation model as a function of the CIE density. It can be seen that GP-CMW becomes very efficient only when the density of CIE events becomes larger that 95%. This suggests that event grouping is a key to lower the simulation time for high CIE density regime and that GPU-based simulations, GP-CMW and GPU-MW, become significantly efficient. For medium CIE density, CMW has the lowest simulation time as it makes use of both GPU and CPU to schedule CIE and FIE events but with lower overhead than that of GP-CMW. However for low CIE density (below 40%), CPU-based simulation proves to outperform GPU-based. It can be seen that the CPU-MW achieves the lowest simulation time with a small gain when compared to CMW.

B. Density Of Recursive Events

A recursive event is an event that is generated due to the execution of anther event during the course of simulation. In this experiment, the recursive events are introduced by varying the wireless channel quality over time. This will increase the probability of losing packets due to wireless error causing the packets to go through the TCP retransmissions, which in this experiment represents the recursive events. Because the

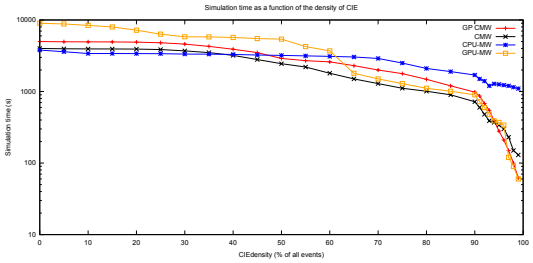


Fig. 13. Impact of CIE density on the performance of the simulation model

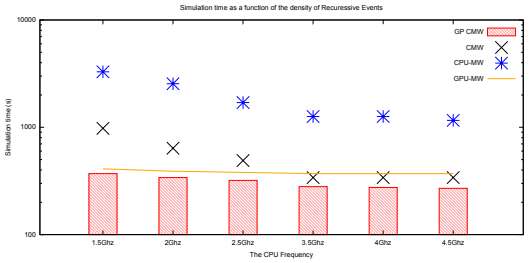


Fig. 15. Impact of CPU frequency on the performance of the simulation model

recursive events concern only FIE, we keep the rate of CIE constant with high density and vary the rate of FIE.

Fig. 14 shows the impact of recursive event density on the simulation time. In can be observed that the GP-CMW model achieves the lowest simulation time for different densities of recursive events. This is mainly due to the event scheduling algorithm of the GP-CMW allowing a rapid execution of recursive events with no causality. While in CMW and MW, a recursive event is re-injected to the event queue as a new event, in GP-CMW, the scheduler attempts to execute such recursive events immediately without reordering them. It has to be mentioned that most of the optimized scheduling algorithms require O(Nlog(N)) to reorder the events in the event queue to retain the simulation consistency, which in turn increases the simulation time.

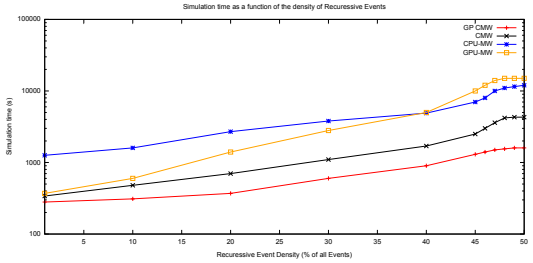


Fig. 14. Impact of recursive event density on the performance of the simulation model

C. Frequency of CPU

In this experiment, we analyze the impact of CPU frequency on the total simulation time. We set the CPUs frequency across the six workstations to be 1.5, 2, 2.5, 3.5, 4 and 4.5 GHz. The simulation scenario evolves 6 ELPs with an identical load at the beginning of the simulation.

It can be observed from Fig. 15 that the GP-CMW not only has the lowest simulation time but also lowest variability as the CPU frequency changes (30% for GP-CMW, 80% for CMW, and 100% for MW-CPU). This flexibility is mainly due to the elasticity of the WU in GP-CMW as each ELP requests WU length as a function of its load. The coordinator may then adapt the length of the global WU based on the performance of each ELP to minimize the overall inactivity time. It is worth mentioning that the most efficient solution to deal with the heterogeneity of compute nodes can be obtained through a load balancing mechanism among ELPs.

VI. Conclusion

Parallel and distributed simulations are considered as the main approach to improve the efficiency for large and extra-large scale simulation. However, existing simulation models do not take into account the heterogeneous computing node architecture combining multi-core CPU with powerful GPUs, which represents key ingredients for a parallel and distributed simulation given a massive number of events. In this context, one of the central challenges is to find an optimal tradeoff between computing, data, and communication locality. At the same time, meta-computing systems composed of several interconnected heterogeneous computing nodes emerge as a real alternative to traditional expensive and complex datacenter. However, to increase the efficiency of such infrastructure, simulation models have to maximize the interactions within each computing node while minimizing the communication and synchronization overhead with other computing nodes.

To address this requirement, we propose a three-level general purpose coordinator-master-worker simulation model. The proposed model is based the commonly used master-worker model and introduces a third top-level process denoted as the coordinator to manage multiple simulation instances in different addressing space. With respect to basic CMW, GP-CMW introduces two optimization layers, namely priority abstraction layer (PAL) to manage the communication across multiple simulation instances, and hardware abstraction layer (HAL) to exploit computing, data, and communication localities. Experimentation results confirm that GP-CMW significantly improves the simulation efficiency in both parallel and distributed context when compared to CMW and MW.

References

Aaby, B., Perumalla, K. and Seal, S. (2010). Efficient simulation of agent-based models on multi-gpu and multi-core clusters, *in Proceedings of the 3rd International Conference on Simulation Tools and Techniques* .

Abdelrazek, A., Kaschub, M., Blankenhorn, C. and Necker, M. (2009). A novel architecture using nvidia cuda to speed up simulation of multi-path fast fading channels, *in Vehicular Technology Conference VTC Spring* .

April, J., Glover, F., Kelly, J. and Laguna, M. (2003). Practical introduction to simulation optimization, *in Simulation Conference*, Proceedings of the 2003 Winter.

Bai, S. and Nicol, D. M. (2010). Acceleration of wireless channel simulation using gpus, *in Wireless Conference, 2010 European IEEE* .

Borries, K., Judd, G., Stancil, D. and Steenkiste, P. (2009). Fpga-based channel simulator for a wireless network emulator, *in Vehicular Technology Conference VTC Spring* .

Chen, L., Huang, J. and Zhang, J. (2012). A latency-hiding algorithm for abms on parallel/distributed computing environment, *in Principles of Advanced and Distributed Simulation, ACM/IEEE/SCS 26th Workshop* .

Cramer, T., Schmidl, D., Klemm, M. and Mey, D. (2012). Openmp programming on intel xeon phi coprocessors: An early performance comparison, *Proceedings of the Many-core Applications Research Community Symposium at RWTH Aachen University*.

Deelman, E., Bagrodia, R., Sakellariou, R. and Adve, V. (2001). Improving lookahead in parallel discrete event simulations of large-scale applications using compiler analysis, *Proceedings of the Fifteenth Workshop on Parallel and Distributed Simulation*, PADS.

Fujimoto, R. (1988). Lookahead in parallel discrete event simulation, *Technical report*. DTIC Document.

Fujimoto, R. M. (1990). Parallel discrete event simulation, *Communications of the ACM* **33**(10): 30–53.

Fujimoto, R. and Nicol, D. M. (1992). State of the art in parallel simulation, *in Proceedings of the 24th conference on Winter simulation* .

Fujimoto, R., Perumalla, K., Park, A., Wu, H., Ammar, M. and Riley, G. (2003). Large-scale network simulation: how big? how fast?, *in Modeling, Analysis and Simulation of Computer Telecommunications Systems* .

Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., Shet, A., Chrysos, G. and Dubey, P. (2013). Design and implementation of the linpack benchmark for single and multi-node systems based on intel xeon phi coprocessor, *in 27th IEEE International Parallel and Distributed Processing Symposium* .

Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*, Elsevier.

Lacage, M. (2010). *Experimentation tools for networking research, Ph*, PhD thesis, UNICE.

Lacage, M. and Henderson, T. R. (2006). Yet another network simulator, *in Proceeding from the 2006 workshop on NS-2: the IP network simulator ACM* .

Liu, J. (2009). *Parallel Discrete-Event Simulation*, Wiley.

Liu, Q. and Wainer, G. (2012). Multicore acceleration of discrete event system specification systems, *Simulation* **88**(7): 801–831.

Nicol, D. M. (1996). Principles of conservative parallel simulation, *in Proceedings of the 28th conference on Winter simulation IEEE Computer Society* .

Park, A. and Fujimoto, R. (2009). Efficient master/worker parallel discrete event simulation, *in Principles of Advanced and Distributed Simulation, 2009 ACM/IEEE/SCS 26th Workshop on* .

Park, A. J. and Fujimoto, R. M. (2012). Efficient master/worker parallel discrete event simulation on metacomputing systems, *IEEE Transactions on Parallel and Distributed Systems* **23**: 873–880.

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

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

Pennycook, S., Hammond, S., Jarvis, S. and Mudalige, G. (2011). Performance analysis of a hybrid mpi/cuda implementation of the naslu benchmark, *ACM SIGMETRICS Performance Evaluation Review* **38**(4): 23–29.

Perumalla, K. (2006). Parallel and distributed simulation: traditional techniques and recent advances, *in Proceedings of the 38th conference on Winter simulationWinter Simulation Conference* .

Perumalla, K. (2009). 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 IEEE Computer Society* .

Quaglia, F. and Cortellessa, V. (2000). Grain sensitive event scheduling in time warp parallel discrete event simulation, *in Proceedings of the 14 workshop on Parallel and distributed simulation IEEE Computer Society* .

Righter, R. and Walrand, J. (1989). Distributed simulation of discrete event systems, *Proceedings of the IEEE 77*, pp. 99–113.

Romdhanne, B. B., Bouksiaa, M. S. M., Nikaein, N. and Bonnet, C. (2013). Hybrid scheduling for event-driven simulation over heterogeneous computers, *in Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation ACM* .

Romdhanne, B. B. and Nikaein, N. (2012). Gpu based simulation testbed for large scale mobile networks, *in Communications and Information Technology, 2012 International Conference on IEEE* .

Romdhanne, B. B. and Nikaein, N. (2013). Coordinator-master-worker model for efficient large scale network simulation, *in International Conference on Simulation Tools and Techniques* .

Romdhanne, B. B., Nikaein, N. and Bouksiaa, M. S. M. (2012). Hybrid cpu-gpu distributed framework for large scale mobile networks simulation, *in Distributed Simulation and Real Time Applications, 2012 IEEE/ACM 16th International Symposium on IEEE* .

Romdhanne, B. B., Nikaein, N., Knopp, R. and Bonnet, C. (2011). Openair-interface large-scale wireless emulation platform and methodology, *in Proceedings of the 6th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks* .

Satish, N., Kim, C., Chhugani, J., Nguyen, A., Lee, V., Kim, D. and Dubey, P. (2010). Fast sort on cpus, gpuss and intel mic architectures, *Technical report*. Technical report, Intel.

Saule, E., Kaya, K. and Catalyurek, U. V. (2013). Performance evaluation of sparse matrix multiplication kernels on intel xeon phi, *Technical report*, arXiv 1302.1078.

Shmueli, E. and Feitelson, D. G. (2003). *Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling*.

Som, T. K. and Sargent, R. G. (1998). A probabilistic event scheduling policy for optimistic parallel discrete event simulation, *in Parallel and Distributed Simulation Proceedings. Twelfth Workshop on, IEEE* .

Wenjie, T., Yiping, Y. and Feng, Z. (2012). A hierarchical parallel discrete event simulation kernel for multicore platform, *Cluster Computing* .

Wolf, M., Cai, Z., Huang, W. and Schwan, K. (2002). Smartpointers: personalized scientific data portals in your hand, *in Supercomputing, ACM/IEEE 2002 Conference IEEE* .

Yoginath, S. B., Perumalla, K. S. and Henz, B. J. (2012). Runtime performance and virtual network control alternatives in vm-based high-fidelity network simulations, *in Proceedings of the Winter Simulation Conference, Winter Simulation Conference* .