Coordinator-Master-Worker Model For Efficient Large Scale Network Simulation

Bilel Ben Romdhanne
Mobile Communication Department, Eurecom
benromdh@eurecom.fr

Navid Nikaein
Mobile Communication Department, Eurecom
nikaeinn@eurecom.fr

Christian Bonnet
Mobile Communication Department, Eurecom
bonnet@eurecom.fr

ABSTRACT

In this work, we propose a *coordinator-master-worker* (CMW) model for medium to extra-large scale network simulation. The model supports distributed and parallel simulation for a heterogeneous computing node architecture with both multi-core CPUs and GPUs. The model aims at maximizing the hardware usage rate while reducing the overall management overhead. In the CMW model, the coordinator is the top-level simulation CPU process that performs an initial partitioning of the simulation into multiple instances and is responsible for load balancing and synchronization services among all the active masters. The master is also a CPU process and provides event scheduling, synchronization, and communication services to the workers. 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 messages passing interface. The worker is the elementary actor of CMW model that performs the simulation routines and interacts with the input and output data, and can be a CPU or a GPU thread.

Compared to existing master-worker models, the CMW is natively parallel and GPU compliant, and can be extended to support additional computing resources. The performance gain of the model is evaluated through different benchmarking scenarios using low-cost publicly available GPU platforms. The results have been shown that the speedup up to 3000 times can be achieved compared to a sequential execution and up to 6 times compared to a mono-GPU MW-based simulation. The hardware activities rate of the CMW services for both CPU and GPU are analyzed in detail.

Categories and Subject Descriptors

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

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.
Simutools 2013, March 05-07, Cannes, France
Copyright © 2013 ICST 978-1-936968-76-3
DOI 10.4108/icst.simutools.2013.251739

Keywords

PADS, PDES, Large scale simulation, System architecture, GPGPU, Heterogeneous computing

1. 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 generally reproducible results, the scalability remains a key challenge. 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 [1].

To speedup a large scale simulation, there are two trivial approaches: either to parallelize and/or distribute the simulation over several instances (also known as logical process LP) and/or to use a dedicated accelerator to handle the bottleneck. The distribution of a simulation over multiple computing nodes delivers a significant scalability gain at the cost of higher complexity and overhead to ensure the simulation correctness and efficiency. The first approach is to use a flat architecture where LPs collaborate to realize the simulation using distributed algorithms for the communication and the synchronization is widely used for small to medium sizes. However it generates a significant management overhead. To limit the overhead in the flat architecture when targeting a large scale simulation, a two level hierarchical architecture was introduced, where a specific process (called the server) ensures the management of the simulation. The involvement of that process varies from one platform to another. The master-worker (MW) model is an example of two-level hierarchical architecture that handles efficiently meta-computing systems [2]. Such a design is optimized for recent public hardware, however specific considerations must be done to increase (i) the data locality as flops are cheap but communication is expensive, and (ii) the simulation efficiency by exploiting the capability of heterogeneous computing node. To deal with the MW limitations while coping with computational challenges of heterogeneous computing node architecture, a hierarchical approaches was proposed in [3]. In that work a new concept based on the interaction between CPU-based and GPU-based component was discussed and a specific consideration for the data locality was introduced nevertheless, it ignore the GPU memory

restriction and induces a constant synchronization delay [4].

In this paper, we propose to consider the meta-computing system, which is composed of several interconnected heterogeneous computing nodes as a system of subsystem. The main rule is to maximize the interactions inside a computing node with minimum communication overhead with outside world. To achieve this, we suggest to extend the master-worker model by introducing a third actor called *coordinator*. The model is denoted as coordinator-master-worker (CMW). At the top level, the coordinator ensures the global time synchronization and the load balancing among the masters. At the second level, the master locally manages the time synchronization and event scheduling among the workers. At the third level, the workers are the executing threads performing tasks. 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). In contrast to hierarchical MW models, in CMW model, event generation process is capable of producing native parallel event sets. Furthermore, the scheduling policy is capable of managing both cloned and independent foreign events. Finally, the synchronization and communication processes target to maximize the locality inside the ELP by applying domain-specific operations between CM and MW. In addition, there is no vector-state exchange between the coordinator and masters, and each master must ensures the simulation correctness independently, thereby the coordinator is not considered as a top-level master as it neither assigns tasks nor transfers data.

The reminder of the paper is organized as follows. In Section 2, we present the background information. Section 3 provides the related work. In Section 4, we present the CMW model and its features. Section 5 presents the benchmarking scenarios and validation results for the CMW model in comparison with MW model. Finally Section 6 concludes the paper and provides a short summary of the contribution of this work.

2. BACKGROUND

This section provides a technological context for the proposed CMW model by highlighting the GPU features and programming model as well as a brief description of Cunetsim network simulator used to benchmark the CMW model.

2.1 Technological Context

Graphics Processing Units (GPUs) are specialized electronic circuits, dedicated to the graphical rendering. Although, their preliminary architecture was rigid they become increasingly programmable, flexible and computationally powerful. Modern GPUs are throughput-oriented devices made up of hundreds of processing cores. They maintain a high throughput and low memory latency by multithreading between thousands of threads. GPUs are based on a two-level hierarchical architecture. They are made of *vector processors* at the top level, also known as streaming multiprocessors (SMs) for NVIDIA GPUs. Each vector processor contains an array of processing cores; termed as scalar processors (SPs). Inside one vector, SPs communicate with each other through an on-chip user-managed memory, termed as shared memory. In addition, the GPU is an independent device which embeds its specific RAM accessible for all processing cores; that qualifies it to be entirely independent. Even if the GPU is promising for high speed parallel computing, the programing model is relatively different than the CPU. However, the concept of General-purpose computing on graphics processing units (GPGPU) aims to provide a user-friendly APIs. In this context, CUDA[5] and OpenCL[6] APIs are the most advanced software programming solutions and share the same SPMD(Single Program Multiple Data) programming model. Due to its developed ecosystems and the wide range of compliant libraries, we choose the CUDA environment as a technological support.

2.2 Cunetsim Framework

Cunetsim is a wireless mobile network simulator design for large scale simulations. The main idea is to perform an entire network simulation inside the GPU context, where each simulated node is executed by one dedicated GPU core, i.e. independent execution environment for each simulated node. Cunetsim implements the MW model and introduces an innovative CPU-GPU co-simulation methodology in that it considers the GPU as a main simulation environment and the CPU as a controller. In addition, nodes communicate using the message passing approach based on the buffer exchange without any global information. It has to be mentioned that Cunetsim is developed following a hardware-software co-design methodology optimized for the NVIDIA GPUs architecture.

In what is relevant to this work, we focus on extending the MW architecture to support an additional actor, the coordinator, for the purpose of benchmarking. In fact, Cunetsim considers the pair CPU-GPU as a unified shared memory parallel system in which the master is a CPU process and workers are GPU threads. The master ensures the correctness and events scheduling of the simulation while each worker is logically associated with one simulated node, where all its data and processes are on the GPU. However, the initial design of Cunetsim only supports one GPU, which limits the scalability of the framework by that of used GPU in terms of available memory and computational power [7]. To resolve the memory size limitation, Cunetsim applies a dynamic off-loading and on-loading of workers between the CPU and GPU context.

3. RELATED WORK

Parallel desecrate event simulators (PDES) are built around the collaboration of several logical processes, where the simulation context is broken-down into several simulation instances with respect to the spatial or functional distribution [8]. To address the scalability issue of PDES, there are three main approaches [9]: architectural optimization, local optimization, and bottlenecks acceleration.

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 [10]. The scalability remains an issue in the flat design when the number of LPs increase [11]. In the literature several optimizations, such as the lookahead [12] and the opportunistic and combined synchronization [9], 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 [13]. 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 nodes). 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 [14]. The number of tiers depends on the setup and available resources, which could potentially cause large synchronization delay due to cascading of the masters. This concept is extended to support GPU [3], where the synchronization and communication overhead is significantly reduced in term of 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 then traditional LPs [15, 4], thus the latency issue needs to be addressed.

The local optimization aims to improve 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 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[16] . 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) [17]. A similar approach, introduced by Park et all [18, 19] 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 [20], 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. A smart usage of that capabilities allows a significant performance gain [21]. Nonetheless, that approach is closely related to the implementation of each solution in one side and to the considered hardware on the other side.

Bottlenecks acceleration aims to improve the simulation efficiency by reducing the impact of a specific part of the simulation which broken-down the system due to its process complexity. Except software solutions which back to the previous case(local optimization), bottlenecks acceleration offloads that process on a specific hardware such as DSPs, FPGAs or GPUs. The DSP is mainly used to process signals and presents a real gain when the simulation considers physical phenomenas such as radio signal simulation, but does not offer a rich programming model suitable for experimentation. The FPGA provides a great tradeoff between the efficiency of the DSP and a reasonable programming flexibility. therefore it was largely used to accelerate existing solutions, in particular, Steenkister et all [22] used the FPGA as a signal accelerator for wireless network simulation. The OpenAirInterface [23] initiative provides an SDR implementation of 4G wireless network (i.e. LTE/LTE-A) using full GPP model, while the RF subsystem is processed by a specific FPGA. Even if FPGA provides an important processing gain, it does not provide a flexible programming model and cannot be used in large scale. In that context, the GPU is an emerging solution which combines programmability, computing power and advanced parallelism capabilities. The advantage of the GPU is also demonstrated by Perumalla et all [24] and several works used it as a channel simulation accelerator [25, 26]. In this context, the GPU becomes a promising solution providing perspectives for large and extra-large scale parallel and distributed simulations. The main limitation of the GPU, however, relies on its programming model, which is not fully compliant with the x86 architecture. 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 [27].

This work presents a new simulation model, denoted as coordinator-master-worker, which aims to increase the efficiency of the large scale network simulation for heterogeneous computing node architecture. The coordinator is introduced as a third-level actor to ensures the master synchronization and load-balancing services. Furthermore, the task of master as defined in the standard MW model is revised to maximize the locality, which is achieved by allowing the master to uniquely manage a pool of workers colocated on the same shared memory context. The worker is a lightweight thread operating on specific computing resource, i.e. GPU or CPU, within the same memory space. Finally, the CMW model guarantees the simulation correctness through a periodic time interval acting as a checkpoint mechanism instead of the state vector transition, which proves to achieve a significant gain in terms of data transfer delay and synchronization.

4. COORDINATOR-MASTER-WORKER ARCHITECTURE

The CMW architecture is designed around three actors: coordinator(C), master(M) and worker(W). The coordinator, is a top-level simulation CPU process with two fundamental tasks: load balancing and synchronization among all the active masters. The master, is also 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 messages passing interface (MPI) [28]. The worker is the elementary actor of CMW that performs the simulation routines and interacts with the input and output data. Typically, a worker can be a CPU or a GPU thread. In CMW, there is only one coordinator operating on N masters, each of which manages K workers. We denote one master and its associated K workers as an extended logical process (ELP).

In the CMW model, the simulation is first distributed

over a certain number of workers (all simulated components in classical terminology) for the considered simulation scenario. Then, workers are partitioned into separated simulation instance according to the user-defined spatial and/or functional policies. Each simulation instance is managed by one master, and all the workers with the same simulation instance interact with the external world uniquely through the master. In a simple case, each simulated instance is performed by one master on one computing nodes. The communication is done through a diffferent dedicated shared memory, as shown in Figure 1, depending on the locality of the workers with respect to each other and the underlying target-specific (i.e. CPU or GPU) memory capabilities.

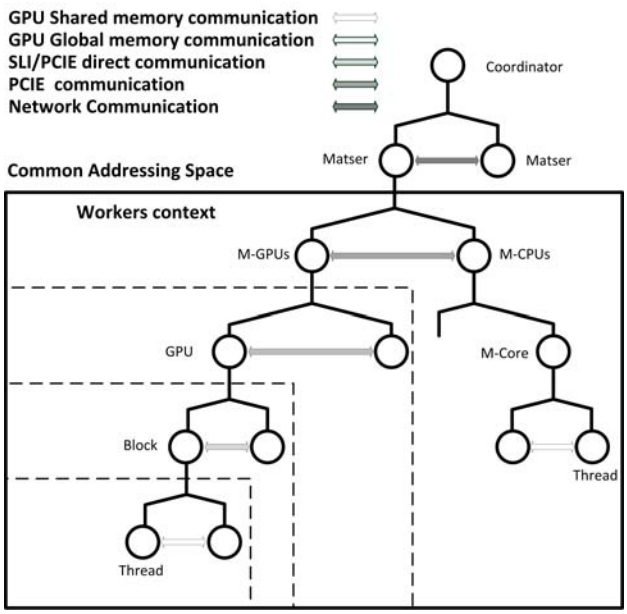


Figure 1: The CMW architecture

In the following subsections, we present the main services and features offered by the CMW model.

4.1 Massive Parallel Events Generation

The massive parallelism concept is a suitable software model for SIMD hardware, and in particular for the GPU programming. The main idea consists of generating cloned threads, each of which performs the same operation on an independent data. This concept is derived from the graphical processing software, where each pixel or polygon is processed independently and in parallel by the same algorithm. We propose to generate similar and independent events rather than detecting events that can potentially be executed in parallel.

4.2 Parallel Events Scheduling

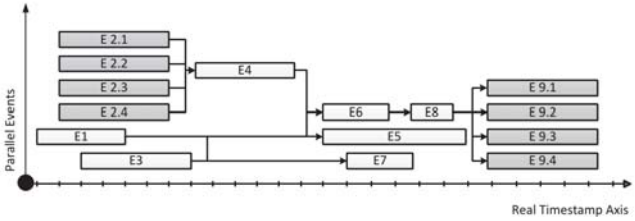
In CMW, the event scheduling is done independently for each ELP by the master, and each master manages locally all its events. The event scheduler is designed based on the hardware and software co-design methodology for CPU and GPU target, and it is performed both at the software and hardware level. At the software level, the scheduler organizes the events in two dimensions, namely in time interval and in parallel event group (or blocks in GPU terminology), which are later used for an efficient hardware mapping. At the hardware level, the embedded GPU GigaThread hardware scheduler first distributes event thread blocks to various SMs, and second assigns each individual thread to an SP inside the corresponding SM (refer to Section 2.1). Furthermore, the scheduler assigns two timestamps to each event: the real timestamp as provided by the event generator used to keep track of event timing, and the execution timestamp as calculated by the event scheduler used to maximize the hardware activity rate without compromising the simulation correctness.

The software scheduler operates as follows. It first *partitions* an arraylist into the time interval and the parallel event group, as shown in the Figure 2(a) and 2(b). Then, it *sorts* all the events in time and regroups those events that can be executed in parallel either into cloned independent events (CIE) if the events differ only in data, or into independent foreign events (IFE) if events differ in both algorithm and data. Finally, it *distributes* events over the arraylist, where each array represents the parallel event group for a given time interval (see Figure 2(c)). 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. To increase the efficiency, when an event is assigned to one interval, its execution timestamp will be aligned with the beginning of that interval. However, if that event has one or multiple dependencies that falls within the same time interval, the scheduler preserves such dependencies and executes the event before the others, otherwise the execution of all the other events will be postponed to the next time interval. The soft scheduler makes use of asynchronous and non-blocking CUDA calls to communicate with the GPU through the driver, which in turn generates the corresponding thread blocks on-the-fly. Upon the reception of thread blocks from the driver, the hardware scheduler dispatches received blocks to available SMs and transfers the data from the CPU to the GPU context. When an SM receives a thread block, the scheduler distributes the wrap of 32 threads executing the same path in the CIE to SPs.

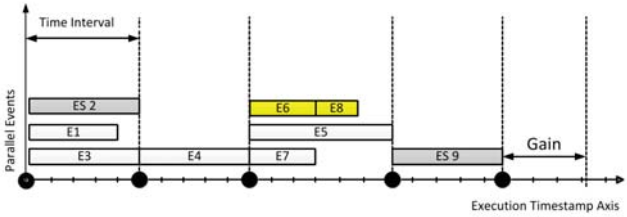
Figure 2 illustrates the event management in CMW software scheduler. We observe that E2.1, E2.2, E2.3 and E2.4 are the CIE and are grouped into a unique entry as an ES2. Events within the same time interval, namely ES2, E1, and E3 are aligned with the beginning of that interval, and that how the event dependency between E6 and E8 is preserved by the scheduler.

4.3 Hierarchical Synchronization Model

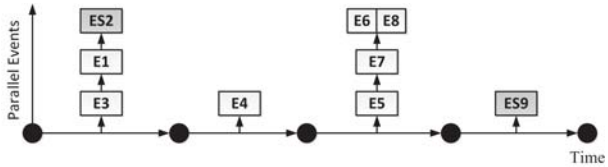
In simulations, there are three distinct notions of time: (i) the physical time, representing the time that a physical event requires to be performed, (ii) the simulation time, representing the physical time in the simulation, and (iii) the wallclock or execution time, which is the elapsed real time during the execution of the simulation, as measured by a hardware clock. In a distributed and parallel setting, we further distinguish two different times depending on whether the simulation is time-driven or event-driven. In the first case, the time is locally represented using an independent clock for each LP, which advances from the current event timestamp to the next one, while in the latter case, the time is represented using a global clock for all LPs, which



(a) Original events graph according the real timestamps



(b) Parallel event groups and the execution timestamps



(c) Software representation the scheduled event arraylist

Figure 2: CMW event scheduling model

advances continually from one value to the next with a predefined granularity.

In the CMW, each actor has its instance of the clock, which are synchronized following a hierarchical model. First, the coordinator defines the duration of a work unit (WU) regarding the simulation time (e.g. 1s), and advances the clock sequentially following a time-driven model from the timestamp of current to the next WU. Then, the clock of each master is synchronized at the beginning of each WU. Each master advances its own clock following a time-driven model from the current to the next execution time interval. Master updates the clock of active workers (i.e. workers executing an event) to the real timestamp of that event. As a result, the clock of workers are updated following an event-driven model and it advances from the real timestamp of the current event to the next event.

The synchronization protocol is performed between the coordiantor and associated masters (C-M) as well as the master and associated workers (M-W). For the C-M synchronization, the coordinator sends an WU message and specifies the WU ID and its duration, i.e. WU(ID,TIME) allowing masters to perform the simulation during the WU time (see Figure 3). When the WU is finished, each master acknowledges the WU by sending an MACK message corresponds to the triple (M_ID, WU_ID, W_ID[]). The latter represents a list of workers' ID used to inform the coordinator about the workers associated with different spatial area (mainly due to mobility), which is used for the purpose of load balancing as described in Section 4.4. During each WU, masters are synchronized based on a lookahead mechanism. For the M-W synchronization, it is performed locally as follows. When receiving a WU message from the coordinator, the master starts the event scheduling and updates the clock of each active worker, before executing any related event, to the real timestamp of that event. Limiting the update procedure only to active workers allows a significant reduction in synchronization overhead without compromising the simulation correctness.

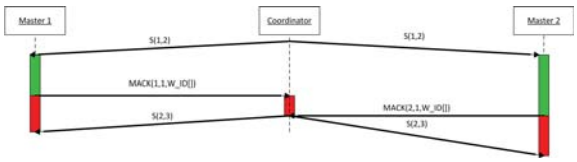


Figure 3: Typical message sequence diagram of the synchronization protocol between the coordinator and the masters

4.4 Load Balancing

In CMW, the load balancing is done by the coordinator and consists of (i) spatial mapping of ELPs to available computing nodes when the simulation starts, and (2) monitoring the efficiency of each master in terms inter-ELP communication during the simulation. The latter may trigger a worker migration between ELPs under the control of the coordinator to maximize the locality of communicating workers during the simulation. Such a migration happens between two work units and may introduce an additional delay in the course of simulation caused by the context transfer and update from the source ELP to the target ELP.

4.5 Hierarchical communication model

The communication model of CMW is based on the message passing, where each message is physically written on the destination buffer. To maximize the efficiency of the workers, two buffers are introduced for each worker in the CMW model: in-buffer and out-buffer. The in-buffer contains all the messages that is subject to be received during the receive event of each worker, while the out-buffer includes all messages that must be sent during the send event of each worker. Depending on the locality of the workers and the underlying target-specific (i.e. CPU or GPU) memory capabilities, the communication is done through a different yet dedicated shared memory (see Figure 1). Four different methods are possible as described 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 writen to the in-buffer of the destinations ensuring that each worker has the entire control on its message.

2. If both workers are in 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 destination will receive distinct message 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 SLI/PCIe 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 foreign ELPs, the source ELP writes the message on a global out-buffer of the master. The master will use the network communication to transfer the message to the destination master.

In what concerns the master, it has three buffers: in-buffer, out-buffer, and one global out-buffer for inter-ELP communication. All the workers within the same ELP can use the global out-buffer to communicate with the remote workers. This global out-buffer is processed by the master periodically, which aggregates incoming messages into different message bundles as a function of their target ELPs. To determine the target ELP, each master maintains a table allowing to look up the target ELP based on the ID of the worker. This table is built when the simulation starts and updated when a migration happens (by the coordinator). Upon the reception of a message bundle by the target ELP, the original messages are restored and delivered to the target worker. This operation introduces an extra delay on message delivery as it does not provide a per-message callback mechanism allowing workers to request a fast service. However, it minimizes drastically the overhead related to the transmission of a large number of messages.

5. EXPERIMENTAL EVALUATION

The testbench to evaluate the CMW model relies on three parallelization frameworks, namely CUDA, OpenMP and MPI, and one development Kit, the PGI suite as shortly explained below.

1. The Compute Unified Device Architecture (CUDA) is the software parallel computing platform and a programming model created by NVIDIA. It provides API, libraries, compilers, debugger, and drivers to manipulate easily NVIDIA GPUs for a general purpose processing [5].

2. The Open Multiprocessing (OpenMP) is an API that supports multiprocessing on a shared memory context. Based on pre-compilation directives, the OpenMP handles the creation and management of threads using fork and join operations.

3. The Message Passing Interface (MPI) is a portable message-passing system providing an abstraction layer for distributed programming, which is usable on a variety of platforms. In particular, it provides the concept of barriers synchronization with a reduced latency and communication overhead [29].

4. The PGI suite is a commercial C/C++ compiler, which provides several automatic parallelization features. Further, it generates optimized binary for different CPU architectures and incorporates a full CUDA CC++ compiler for targeting X64 CPUs. It supports the unified binary technology, which consists on the creation of a multi-target binary (GPU, INTEL CPU and AMD CPU) from an initial native CUDA code.

In the following subsections, we present the benchmarking scenario and the experimental setup. Then, we highlight the evaluation metrics and performance evaluation with the relevant simulators. Finally, we analyze and discuss the results.

5.1 Scenario and Setup

To evaluate the performance of the CMW model compared to a standard MW model under large scale conditions, we extend the benchmarking methodology proposed in [7] to support distributed scenario. The benchmarking scenario is illustrated in Figure 4 and is based on a static network topology, where nodes are arranged into six independent activity areas of $5km^2$, named AA0 to AA05. In each AA, nodes are placed within a square. By default, the last node of each AA is connected to the first node of the next square. The scenario includes one traffic source, which generates 600 uniformly distributed packets with packet size if 128 bytes and inter-departure time of 1 second. The generator is located at the AA0 and the destination is located on the AA05. All nodes forward unseen packets after a delay of 1 second, thus flooding the entire network. The network connectivity is modeled using a simple probabilistic dropping probability, which is identical for all links. The sender is the node with the lowest ID and the receiver is the one with the highest ID.

This benchmarking scenario has the particularity that the total number of events increases linearly as a function of the number of nodes in the network. Thus, it allows a proper analysis of both the event scheduler and the synchronization process.

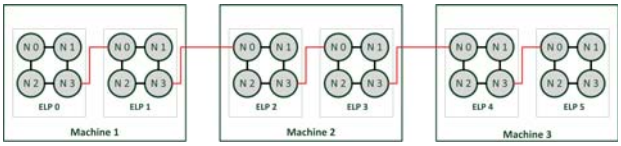


Figure 4: Benchmarking scenario with six distinct activity areas

The hardware setup is based on a meta-computing nodes interconnected through a Gigabit local area network. It includes 4 workstations: three of them include each an INTEL i7 3930k CPU (6 cores with hyper-threading), 32GB of DDR3 and Two GeForce GTX680 2GB (1536 cores for GPGPU computing), and the last node includes an i7 930 (4 cores) with one GTX460 and plays the role of the coordinator in case of CMW architecture. The OS is Ubuntu Linux 11.10, the PGI compiler version is the 12.9 and the Nvidia driver version is 295.41. In the present study, we compare the performance of the following simulators:

1. **D-NS3:** is the distributed CPU-based version of NS3 based on the MPI API that implements standard flat model. The software setup of D-NS3 is composed of 6 LPs each of which simulates one AA and each LP uses 3 cores of the CPU.

2. **Cunetsim:** is the original version of the Cunetsim simulator with mono-GPU support based on the MW model [7]. The software setup of Cunetsim is composed of 1 ELP, which simulates 6 AA using one GPU.

3. **D-Cunetsim-GPU:** is the distributed multi-GPU version of the Cunetsim that implements the CMW model. It includes 6 GPUs and 3 computing nodes. The software setup is composed of 6 ELPs each of which simulates one AA and each ELP uses one GPU.

4. **D-Cunetsim-CPU:** is the distributed CPU version of the Cunetsim that implements the CMW model. It also has 6 GPUs and 3 computing nodes. The software setup is composed of 6 ELPs each of which simulates one AA and each ELP uses 3 cores of one CPU.

We consider two fundamental metrics: the relative speedup[1] and hardware activity or usage rate measured in percentage of total execution time. The relative speedup of a distributed network simulator is defined as the ratio between the runtime of a sequential reference simulator, standard NS3 in over that of the benchmarking simulator, which provides an overall system efficiency regardless of the hardware. The hardware activity rate is a monitoring metric provided by the OS representing the percentage of time used by the components of the CMW model.

5.2 The Relative Speedup

The relative speedup is a representative metric used to measure the efficiency of a parallel and distributed simulators. However there are two main parameters, which influence directly the speedup: (i) the size of each simulated area and (ii) the inter-area (ELP in case of CMW) communication.

5.2.1 Impact of network density

To study the impact of the network density in simulation speedup, we select four representative network scales, including small, medium, large, and extra-large, each of which contains receptively 150, 2.4k, 15K and 303k nodes placed uniformly across the 6 AA. Results are summarized in Figures 5 for each network scale indexed from 1 to 4, where we observe four main conclusions. First, the cost of initialization for all the distributed simulations is extremely high for a small scale simulation. We observe that for 150 nodes, the D-NS3 and D-Cunetsim-CPU are only two times faster than the benchmarking sequential reference simulator, NS3, while they used 6 cores. As for the Cunetsim and D-Cunetsim-GPU, they are slower than NS3 as the data transfer delay between the CPU and GPU context becomes dominant while the GPU processing delay is negligible.

For a medium scale network (2.4k nodes on the simulation which represents 400 nodes per AA while we have 1536 cores), the normalized speedup becomes significant and GPU-based simulators can perform the simulation up-to 8 times faster than that of the CPU-based. The D-NS3 and D-Cunetsim-CPU present a gain of 9 and 10.8 times with respect to NS3. It has to be mentioned that this scenario includes a small number of independent events, which in turn reduces the scheduling time. Moreover, results of the Cunetsim and the D-Cunetsim-GPU remain below the expectations as GPUs are extremely under-utilized (400 workers per AA).

From 15K nodes, the event density increases rapidly and the scheduling becomes an expensive task. In particular, its impact is more significant on the performance of CPU based simulator. Indeed, the *per events scheduler* used in the D-NS3 proved to be less efficient by a factor of 2 than *per events-set scheduler* used in D-Cunetsim-CPU (see section 4.2). As for the scalability, Cunetsim proved to be scalable with a relative speedup of 380 times. We observe however, that the performance of D-Cunetsim-GPU is significantly improved (the speedup is about 1000 times). In this scenario, the maximum capacity of the GPU is not yet reached.

To simulate 303K nodes, each AA is populated with 50K nodes, and the event rate becomes extremely large, making per-event scheduling very costly. In fact each D-NS3 instance must schedule about 121M events during the simulation while the D-Cunetsim-CPU schedules 10K of events-set. Therefore, when D-Cunetsim-CPU reaches a speedup of 17.8 times, D-NS3 achieves only a speedup of 7 times while both use the same hardware. We also found that Cunetsim achieves the maximum gain, speedup of 560 times, on such a large scale scenario as the GPU reaches its optimal operating level with the total number of 50k threads [30]. Furthermore, D-Cunetsim-GPU presents a significant speedup of 3300 times. We note that D-Cunetsim-GPU outperforms Cunetsim in the order of 5.89, which is very close to the number of GPU used in each case, i.e. 1:6. By comparing Cunetsim and D-Cunetsim-GPU, we conclude that the overhead due to the synchronization and the inter-ELP process remains reasonable in the CMW model.

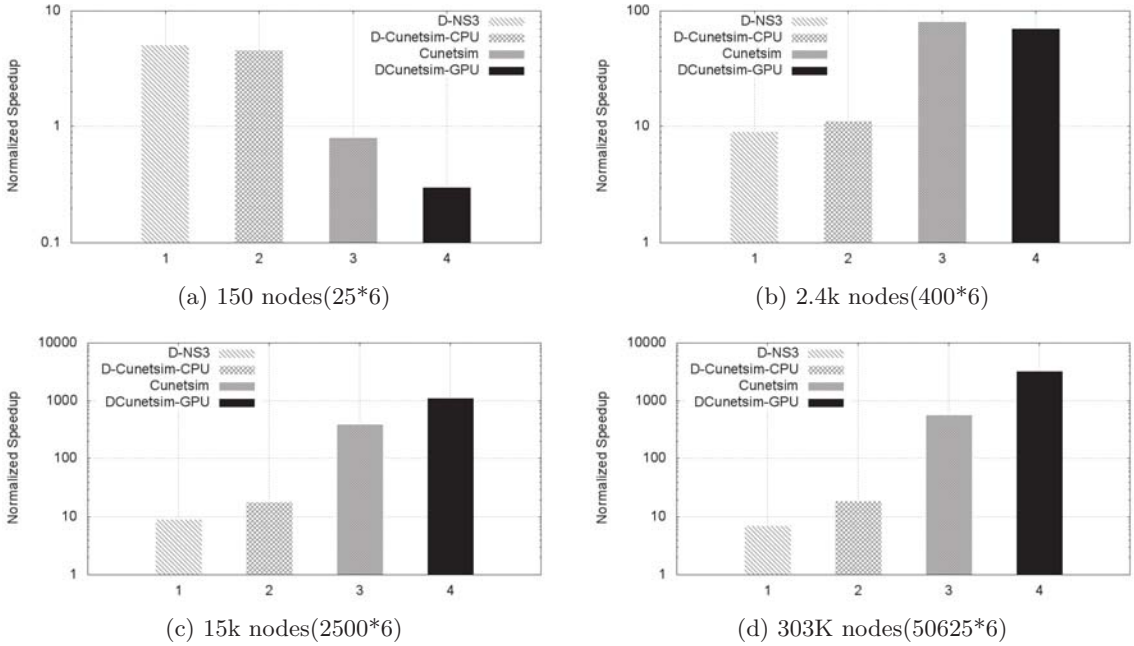
5.2.2 Impact of inter-AA communication

To study the impact of the inter-AA communication, we increase the inter-connection at the level of border nodes. We fix the number of node to 303K nodes and we increase the inter-connection from 5% to 50% with intermediate value of 10% and 25%. Figure 6 summarizes corresponding results. The first point that draws our attention is that the performance of the CPU based frameworks remains stable as the number of inter-AA connection increases. The speedup of D-Cunetsim-CPU decreases by 25% and that of D-NS3 by 37% when the inter-AA connection increases from 1% to 50%. The higher robustness of the CMW model compared to the traditional MW model is essentially due to the hierarchical communication model (see section 4.5). The measurement demonstrates that the D-Cunetsim-CPU model requires in average 23% less network bandwidth than the D-NS3 while the total number of exchanged message is equivalent. The second relevant point is that the performance of the Cunetsim (Mono GPU based on the MW model) also remains stable as the number of the inter-AA connection increases. Its relative speedup decreases from 560 to 537 (5%). This stability is due to the fact that nodes communicate always using CUDA memory primitives regardless of their AA. Concerning D-Cunetsim-GPU framework, we observe that its efficiency is strongly correlated with the communication rate between different ELPs. Thus, the more ELPs messages exchanged, the less efficient the system becomes. The relative speedup decreases from 2900 to 300 times. This experimentation demonstrates the importance of the scenario design and the load balancing capabilities when dealing with the distributed simulation.

5.3 Hardware Activity Rate

In Figure 7, we summarize the hardware activity rate of

[1]Relative speedup and normalized speedup are used interchangeably



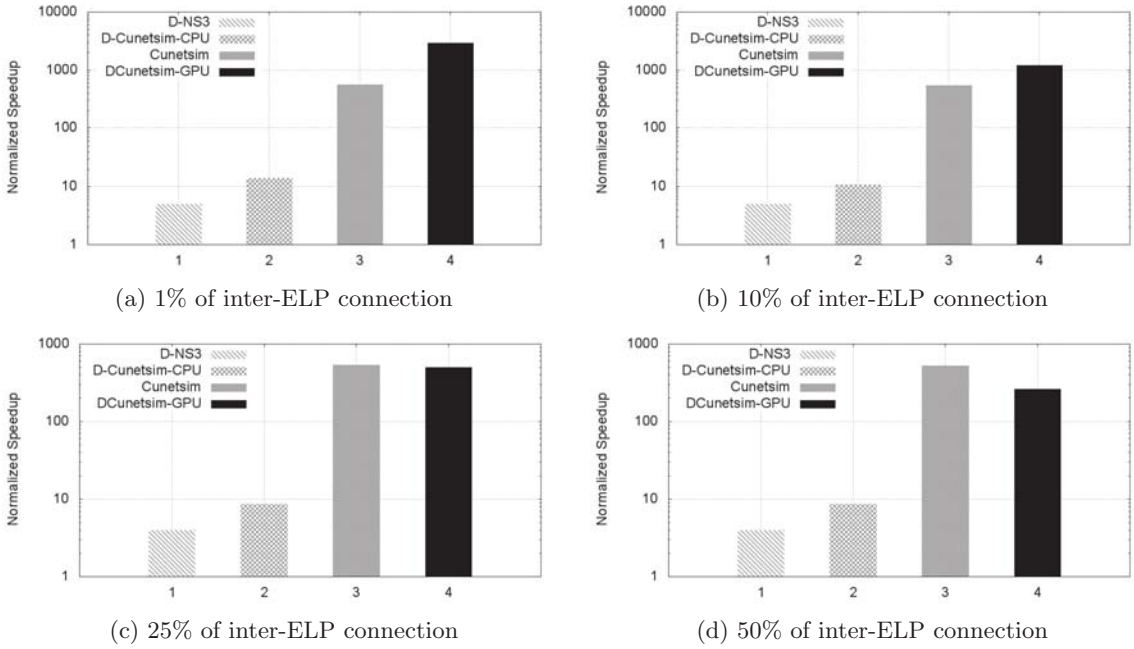
(a) 150 nodes(25*6)

(b) 2.4k nodes(400*6)

(c) 15k nodes(2500*6)

(d) 303K nodes(50625*6)

Figure 5: The normalized speedup of different simulators as a function of network density



(a) 1% of inter-ELP connection

(b) 10% of inter-ELP connection

(c) 25% of inter-ELP connection

(d) 50% of inter-ELP connection

Figure 6: The normalized speedup of different simulators as a function of the percentage of inter-AA connections

(a) CPU activity rate (%)

(b) GPU activity rate (%)

Figure 7: Hardware activity rate of both CPU and GPU as a function of the network size

the CMW software components for both CPU and GPU. For the CPU target, in addition to idle time and waiting time, we measure three activities as follows. First, the synchronization and communication time (Sync. & com.) of CMW provided by MPI service. Second, the event scheduling time capturing the event management and assignment at the software level. Third, the data transfer time representing the CPU to GPU context transfer. We consider 4 networks scales as described in Section 5.1. The major observation is that during a large scale experimentation (case 4), the CPU is under-utilized (less than 10%), and that some additional tasks such as real time monitoring can still be performed by the CPU without compromising the simulation efficiency. Furthermore, we observe that the scheduler uses less than 4% of the CPU, which presents an exceptional efficiency level for such a large scale scenario.

For the GPU target, in addition to the idle time, we measure two other activities: (1) the processing time representing the time required to execute events, and (2) the data transfer time representing the GPU to CPU context transfer. The idle time is measured as the average of the idle time of all GPU cores. We observe that the usage of the GPU, is inversely proportional to that of the CPU. Thus, for a large scale experimentation the GPU is heavily loaded with negligible idle time. We can conclude that the CMW model is more adequate for large scale simulations with the heterogeneous computing node architecture.

6. CONCLUSIONS

The network simulation becomes a main support of the technological development life cycle. Even if simulations provide generally reproducible results, the scalability remains a key challenge. In fact, increasing the scale of the simulated system leads to a nonlinear increase of the required resources on one hand and the execution time on the other hand, which in turn reduces significantly the simulation efficiency [1]. Parallel and distributed simulations is considered as the main approach to improve the speedup for large and extra-large scale simulation. However, existing simulation models does not take into account the heterogeneous computing node architecture combining multi-core CPU with powerful GPUs, which represents a key ingredients for a parallel and distributed simulation in view of large number of events. In this context, one of the main challenges is to find an optimal tradeoff between the communication and data locality.

In this work, we propose to consider the meta-computing system, which is composed of several interconnected heterogeneous computing nodes as a system of subsystem. The proposed concept is to maximize the interaction within each computing node while minimizing the communication overhead among. To achieve this, we suggest to extend the master-worker model to support heterogeneous computing node architecture by introducing a third actor called coordinator . The model is denoted as *coordinator-master-worker (CMW)*. At the top level, coordinator ensures the time synchronization and the load balancing among the masters. At the second level, the master locally manages the time synchronization and event scheduling among the workers. At the third level, the workers are the executing threads performing tasks. From the coordinator point of view, the master manages one simulation instance, which is why the master-worker subsystem is refereed as extended logical process (ELP). In contrast to existing hierarchical software model, the CMW model is based on the hardware and software co-design methodology, where event generation and scheduling processes are natively parallel and compliant wih the SIMD architecture of GPUs. In addition, there is no vector-state exchange between the coordinator and masters, and each master ensures the simulation correctness independently, thereby the coordinator is not considered as a top-level master as it neither assigns tasks nor controls data.

Comparative evaluations prove that the CMW model provides a significant gain in term of speedup compared to a MW on the same hardware platform. In particular, the gain is more significant when targeting large to extra-large scale simulation. The results have been shown that the speedup up to 3000 times can be achieved compared to a sequential execution and up to 6 times compared to a mono-GPU MW-based simulation. The analysis of the hardware activities rate have shown that the CMW is capable of maximizing the hardware usage.

Acknowledgment

The research leading to these results has received funding from the European Research Council under the European Community Seventh Framework Programme (FP7/2007- 2013) n° 248993 (LOLA) and n° 257616 (CONECT).

7. REFERENCES

[1] B. Aaby, K. Perumalla, and S. 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] A. Abdelrazek, M. Kaschub, C. Blankenhorn, and M. Necker. A novel architecture using nvidia cuda to speed up simulation of multi-path fast fading channels. In *Vehicular Technology Conference, 2009. VTC Spring 2009. IEEE 69th*, pages 1–5. IEEE, 2009.

[3] J. April, F. Glover, J. Kelly, and M. Laguna. Practical introduction to simulation optimization. In *Simulation Conference, 2003. Proceedings of the 2003 Winter*, volume 1, pages 71–78. IEEE, 2003.

[4] S. Bai and D. Nicol. Acceleration of wireless channel simulation using gpus. In *Wireless Conference (EW), 2010 European*, pages 841–848. IEEE, 2010.

[5] B. Bilel and N. Navid. Cunetsim: A gpu based simulation testbed for large scale mobile networks. In *Communications and Information Technology (ICCIT), 2012 International Conference on*, pages 374–378. IEEE, 2012.

[6] B. Bilel, N. Navid, K. Raymond, and B. Christian. Openairinterface 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*, pages 109–112. ACM, 2011.

[7] M. S. M. B. Bilel Ben Romdhanne, Navid Nikaein. Hybrid cpu-gpu distributed framework for large scale mobile networks simulation. In *16th IEEE/ACM DS-RT :The International Symposium on Distributed Simulation and Real Time Applications*. Ieee/ACM, 2012.

[8] K. Borries, G. Judd, D. Stancil, and P. Steenkiste. Fpga-based channel simulator for a wireless network emulator. In *Vehicular Technology Conference, 2009. VTC Spring 2009. IEEE 69th*, pages 1–5. IEEE, 2009.

[9] L. Chen, J. Huang, and J. Zhang. A latency-hiding algorithm for abms on parallel/distributed computing environment. In *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pages 187–189. IEEE, 2012.

[10] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. Openmp programming on intel r xeon phi tm coprocessors: An early performance comparison. 2012.

[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] S. Green. Particle simulation using cuda. *NVIDIA Whitepaper, December 2010*, 2010.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message passing interface*, volume 1. MIT press, 1999.

[15] G. Kunz. Parallel discrete event simulation. *Modeling and Tools for Network Simulation*, pages 121–131, 2010.

[16] N. Ladas. Programming the gpu using opencl introductory tutorial.

[17] J. Liu. *Parallel Discrete-Event Simulation*. Wiley Online Library, 2009.

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

[19] A. Park and R. Fujimoto. Efficient master/worker parallel discrete event simulation. In *Principles of Advanced and Distributed Simulation, 2009. PADS'09. ACM/IEEE/SCS 23rd Workshop on*, pages 145–152. IEEE, 2009.

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

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

[22] 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.

[23] D. Patterson. The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges. *NVIDIA Whitepaper*, 2009.

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

[25] 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.

[26] K. 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.

[27] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[28] R. Righter and J. Walrand. Distributed simulation of discrete event systems. *Proceedings of the IEEE*, 77(1):99–113, 1989.

[29] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey. Fast sort on cpus, gpus and intel mic architectures. Technical report, Technical report, Intel, 2010.

[30] T. Wenjie, Y. Yiping, and Z. Feng. A hierarchical parallel discrete event simulation kernel for multicore platform. *Cluster Computing*, pages 1–9, 2012.