

Large-scale network simulation over heterogeneous computing architecture

Bilel Ben Romdhanne

▶ **To cite this version:**

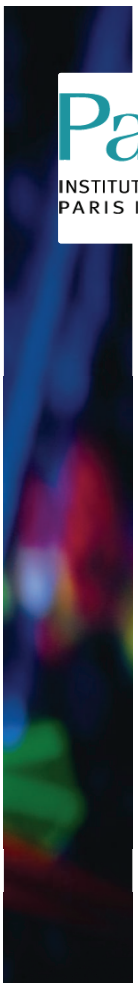
Bilel Ben Romdhanne. Large-scale network simulation over heterogeneous computing architecture. Other [cs.OH]. Télécom ParisTech, 2013. English. NNT : 2013ENST0088 . tel-01151414

HAL Id: tel-01151414
https://pastel.hal.science/tel-01151414

Submitted on 12 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2013-ENST-0088

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

Télécom ParisTech

présentée et soutenue publiquement par

Bilel BEN ROMDHANNE

le 16 Décembre 2013

Simulation des Réseaux à Grande Echelle sur les Architectures de Calculs Hétérogènes

Directeurs de thèse : **Navid NIKAEIN** et **Christian BONNET**

Jury

M. Thierry TURLETTI, DR, Diana, INRIA Sophia Antipolis	Président
M. Kaylan S. PERUMALLA, Professeur adjoint, School of CSE, Georgia Tech	Rapporteur
M. Walid DABBOUS, DR, Diana, INRIA Sophia Antipolis	Rapporteur
Mme. Elisabeth BRUNET, Professeur assistant, G2, Telecom SudParis	Examinateur
Mme. Margaret L.LOPER, Directeur chercheur, ICL, Georgia Tech Research Institut	Examinateur
M. Christian BONNET, Professeur, Communications mobiles, Eurecom	Directeur
M. Navid NIKAEIN, Professeur assistant, Communications mobiles, Eurecom	Directeur

T H È S E

Télécom ParisTech
Ecole de l'Institut Télécom – membre de ParisTech
46, rue Barrault – 75634 Paris Cedex 13 – Tél. + 33 (0)1 45 81 77 77 – www.telecom-paristech.fr

TELECOM PARISTECH

École Doctorale STIC

Sciences et Technologies de l'Information et de la Communication

THÈSE

pour obtenir le titre de

Docteur en Sciences

de TELECOM ParisTech

Mention Informatique

présentée et soutenue par

Bilel BEN ROMDHANNE

Simulation des Réseaux à grande Échelle sur les architectures de calcules hètèrogénes

Thèse dirigée par Navid NIKAEIN et Christian BONNET

Laboratoire EURECOM, Sophia Antipolis

soutenue le 16 Decembre 2013, devant le jury composé de:

Président du Jury	Thierry TURLETTI	, France
Rapporteurs	Kaylan S. PERUMALLA	, USA
	Walid DABBOUS	, France
Examinateurs	Elisabeth BRUNET	, France
	Margaret L.LOPER	, USA
Directeur de thèse	Christian BONNET	, France
	Navid NIKAEIN	, France

© 2013
Ben Romdhanne Bilel
ALL RIGHTS RESERVED

Acknowledgments

I think that one of my major motivation during the last three years, is my dream to write the following: I dedicate this work to my mother that supported, encouraged, powered and resourced me during too many years (thirteen exactly). A special dedication for my father also which gives me the example for a respectable life. I hope that this work will be the pride of my parents. I want to thank my lovely wife which shares with me the same experience, and sharing two PhD at the same time in the same house, is not easy at all! Finally, I hope that one day, I will read a similar dedication written by my young son Ahmed Yessine.

Summary

Large-scale network simulation over heterogeneous Computing architecture

The simulation is a primary step on the evaluation process of modern networked systems. The scalability and efficiency of such a tool in view of increasing complexity of the emerging networks is a key to derive valuable results. The discrete event simulation is recognized as the most scalable model that copes with both parallel and distributed architecture. Nevertheless, the recent hardware provides new heterogeneous computing resources that can be exploited in parallel.
The main scope of this thesis is to provide a new mechanisms and optimizations that enable efficient and scalable parallel simulation using heterogeneous computing node architecture including multicore CPU and GPU. To address the efficiency, we propose to describe the events that only differs in their data as a single entry to reduce the event management cost. At the run time, the proposed hybrid scheduler will dispatch and inject the events on the most appropriate computing target based on the event descriptor and the current load obtained through a feedback mechanisms such that the hardware usage rate is maximized. Results have shown a significant gain of 100 times compared to traditional CPU based approaches.
In order to increase the scalability of the system, we propose a new simulation model, denoted as general purpose coordinator-master-worker, to address jointly the challenge of distributed and parallel simulation at different levels. The performance of a distributed simulation that relies on the GP-CMW architecture tends toward the maximal theoretical efficiency in a homogeneous deployment. The scalability of such a simulation model is validated on the largest European GPU-based supercalculator the TGCC Curie with 1024 LPs each of which simulates up to 1 million nodes.
To further validate the efficiency and scalability of the proposed mechanisms and optimizations, we applied the event grouping, hybrid scheduling, and GP-CMW to popular network simulator $NS-3$. Results have demonstrated a gain of25 times under a large scale deployment including 288 GPUs and 1152 CPU on the TGCC Curie.

Résumé

Simulation des Réseaux à grande échelle sur les architectures de calcules hètèrogénes

La simulation est une étape primordiale dans l' évolution des systèmes en réseaux. L' évolutivité et l' efficacité des outils de simulation est une clef principale de l' objectivité des résultats obtenue, étant donné la complexité croissante des nouveaux des réseaux sans-fils. La simulation a évÃ¨nement discret est parfaitement adéquate au passage à l' échelle, cependant les architectures logiciel existantes ne profitent pas des avancées récente du matériel informatique comme les processeurs parallèle et les coprocesseurs graphique.
Dans ce contexte, l'objectif de cette thèse est de proposer des mécanismes d'optimisation qui permettent de surpasser les limitations des approches actuelles en combinant l'utilisation des ressources de calcules hétérogéne. Pour répondre à la problématique de l'efficacité, nous proposons de changer la représentation d'événement, d'une représentation bijective (évènement-descripteur) à une représentation injective (groupe d'évènements-descripteur). Cette approche permet de réduire la complexité de l'ordonnancement d'une part et de maximiser la capacité d'exécuter massivement des événements en parallèle d'autre part.
Dans ce sens, nous proposons une approche d'ordonnancement d'événements hybride qui se base sur un enrichissement du descripteur pour maximiser le degré de parallélisme en combinons la capacité de calcule du CPU et du GPU dans une même simulation. Les résultats comparatives montre un gain en terme de temps de simulation de l'ordre de 100x en comparaison avec une exécution équivalente sur CPU uniquement.
Pour répondre à la problématique d'évolutivité du système, nous proposons une nouvelle architecture distribuée basée sur trois acteurs logiciels : Un coordonnateur, un master et un worker. Cette architecture réponde simultanément aux limitations des architectures parallèles et distribuées. Nous proposons aussi une version optimisée de l'architecture qui maximise la stabilité et réduit la latence de la communication. Les expérimentations de validation ont était sur le supercalculateur hybride TGCC Curie avec 1024 instance, chacune simule un million de noeuds.
Pour proposer une valider grand-publique de notre approche, nous avons apportés des modifications au simulateur de réseaux NS-3, les modifications portent sur le générateur, l'ordonnanceur et l'exécuteur des évènements.
Les résultats comparatifs apportent la preuve que l'approche proposé ouvre des nouveaux horizons de passage à l'échelle. Nous avons pu atteindre un gain de 25x en combinant le groupement des évènements avec l'ordonnancement hybride.

Contents

1 Introduction **3**
- 1.1 Motivation and Objectives . . . 3
- 1.2 Contributions Storyline . . . 4
- 1.3 Thesis Structure . . . 6
- 1.4 Publications . . . 7
- 1.5 Project Deliverables . . . 8

I Background **9**

2 Network Experimentation **11**
- 2.1 Introduction . . . 11
- 2.2 Network Experimentation Tools . . . 11
  - 2.2.1 Real world Field Trial . . . 11
  - 2.2.2 Real world Testbed . . . 12
  - 2.2.3 Emulation Testbed . . . 13
  - 2.2.4 Simulation Testbed . . . 13
  - 2.2.5 Hybrid Testbed . . . 14
- 2.3 Network Experimentation Aspects . . . 15

3 Discrete Event Simulation **19**
- 3.1 Introduction . . . 19
- 3.2 Discrete-Event Simulation . . . 19
  - 3.2.1 Terminology and Components . . . 19
  - 3.2.2 The Principle . . . 20
- 3.3 Parallel Discrete Event Simulation . . . 20
  - 3.3.1 Discrete Event Simulation Limits . . . 21
  - 3.3.2 Principles of Parallel Discrete Event Simulation . . . 21
  - 3.3.3 Parallel Simulation Model and Algorithms . . . 22

4 Hardware Trends **25**
- 4.1 Introduction . . . 25
- 4.2 Evolution of Computing Chips . . . 25
  - 4.2.1 CPU: Historical Evolution and Trends . . . 25
  - 4.2.2 GPU: Historical Evolution and Trends . . . 26
  - 4.2.3 Emerging Solutions . . . 27
  - 4.2.4 Multi-Core accelerator . . . 27
- 4.3 Parallel Programming: Models and API . . . 28
  - 4.3.1 Pthreads . . . 28
  - 4.3.2 OpenMP . . . 28

4.3.3 MPI . . . 29
4.3.4 CUDA . . . 29
4.3.5 OpenCL . . . 29

5 Related Work 31
5.1 Introduction . . . 31
5.2 Large Scale Simulation . . . 31
5.3 PDES Issues . . . 33
5.3.1 Data Representation . . . 33
5.3.2 Event Scheduling . . . 34
5.4 Considerations Heterogeneous Computing Considerations . . . 34

II Contributions 39

6 Cunetsim: An Experimentation Framework to Discover Scalability Horizons 41
6.1 Introduction . . . 41
6.2 Fundamental Concepts . . . 42
6.2.1 The Worker Pool . . . 43
6.2.2 Separation Between an Event and Its Description . . . 43
6.2.3 Massive Parallel Event Generation . . . 43
6.3 Cunetsim Software Architectures . . . 44
6.3.1 The Worker Design . . . 44
6.3.2 The Master Design . . . 46
6.3.3 Legacy Architecture for Multi-Core CPU . . . 47
6.4 Comparative Performances Results . . . 48
6.4.1 Simulation Runtime . . . 50
6.5 Technical Challenges of GPU-based Simulation . . . 53
6.5.1 Synchronization Challenge . . . 53
6.5.2 Memory Management Challenge . . . 55
6.5.3 Precision Issue . . . 56
6.6 Configuration Issues of GPU-Oriented Simulation . . . 56
6.6.1 Space Representation and partitioning . . . 56
6.6.2 Tuning Parameters: Block Size as a Study Case . . . 56
6.7 Conclusion . . . 58

7 Hybrid Events Scheduler 59
7.1 Introduction . . . 59
7.2 The Hybrid Scheduler . . . 61
7.2.1 Model and Components . . . 61
7.2.2 Scheduling Algorithms . . . 62
7.3 Performance Evaluation . . . 67
7.3.1 Scenario & Setup . . . 68
7.3.2 Comparative Evaluation . . . 69

7.3.3 Performance Analysis 70
7.4 Related Work 74
7.5 Discussion 75
7.6 Conclusion 76

8 General Purpose Coordinator-Master-Worker Model 79
8.1 Introduction 79
8.2 The General Purpose Coordinator-Master-Worker Model 81
8.2.1 Events Management: Description, Scheduling and Execution . 82
8.2.2 The Synchronization Mechanism of the GP-CMW Model . . . 87
8.2.3 GP-CMW Communication Model 88
8.3 Comparative Evaluation 90
8.3.1 Comparative Performance Evaluation 91
8.3.2 Inherent Performance Evaluation 93
8.4 Related Work 101
8.5 Conclusion 103

9 Study Case of PADS Methodology Deployment: NS-3 105
9.1 Introduction 105
9.2 Overview 106
9.3 Events scheduling on NS-3 107
9.4 NS-3 Events scheduler extensions 109
9.4.1 The Explicit CPU Parallelism 110
9.4.2 The Implicit CPU Parallelism 111
9.4.3 The GPU Offloading 112
9.4.4 The Co-scheduler Approach 113
9.5 Comparative evaluation 115
9.5.1 Medium Load 115
9.5.2 High Load 116
9.6 Conclusion 117

III Conclusion 119

10 Conclusion 121

A Experimentation Methodology 123
A.1 Introduction 123
A.2 Scientific Experimentation 124
A.3 OpenAirInterface Experimentation methodology 124
A.3.1 OpenAirInterface Formal Experimentation Methodology . . . 125
A.3.2 Methodology Implementation 126
A.4 Conclusion 127

B Résumé Étendue 129

Bibliography 151

List of Figures

6.1 Mobility Events pattern . . . 45
6.2 Simplified master/workers model that targets GPU execution. . . . 47
6.3 Simplified master/workers model that targets CPU execution. . . . 48
6.4 Simple Grid Topology . . . 49
6.5 End-to-End Packet Loss: . . . 50
6.6 Simulation runtime of the static network: . . . 51
6.7 Simulation runtime of the mobile network: . . . 52
6.8 Block size impact . . . 57

7.1 Events reordering and storage: top figure schematizes an events dependency diagram while bottom figure schematizes how they will be reordered and stored. Events dependency is transformed to interval. 62
7.2 Event scheduler model . . . 63
7.3 Statistical table used for recalibration . . . 67
7.4 Topology of the benchmarking scenario . . . 72
7.5 Normalized speedup with respect to the sequential runtime . . . 72
7.6 The hardware usage rate. . . . 72
7.7 The scheduling cost. . . . 73
7.8 Variation of the input rate vs Time . . . 73
7.9 Output event rate of different algorithms. . . . 73
7.10 Average decision path length during the simulation. . . . 73
7.11 Average interval length during the simulation. It closely reflects the events dependency. . . . 74

8.1 The GP-CMW blocks diagram: . . . 82
8.2 Events Life Cycle . . . 84
8.3 Examples of events grouping: . . . 85
8.4 The simulation runtime of the studied simulators: . . . 92
8.5 The synchronization delay as a function of the number of LPs . . . 93
8.6 The impact of the PAL and HAL on the simualtion runtime: . . . 96
8.7 The synchronization delay as a function of the scenario and the configuration . . . 97
8.8 The average external communication latency: . . . 98
8.9 The average internal communication latency: . . . 99
8.10 The average CPU usage rate: . . . 100
8.11 The average GPU usage rate: . . . 100
8.12 The average RAM usage rate: . . . 100
8.13 The average GRAM usage rate: . . . 101

9.1 Default NS-3events scheduling approach: . . . 107

9.2 Distributed event scheduling approach in NS3: 108
9.3 Scheduling Cost as a function of nodes number 110
9.4 Explicit CPU parallelism: 111
9.5 Implicit CPU parallelism: 112
9.6 GPU offloading: 113
9.7 CO-scheduling approach: 114
9.8 Simualtion runtime of each configuration as a function of the number of CPU cores (1000 nodes) 117
9.9 Simualtion runtime of each configuration as a function of the number of CPU cores (500K nodes) 118

A.1 Experimentation Workflow 125
A.2 The user experimentation workflow: 127

List of Tables

2.1 Characteristics of exciting experimentation tools 17

7.1 List of different scheduling approaches 69

Acronyms

Here are the main acronyms used in this document. The meaning of an acronym is usually indicated once, when it first appears in the text.

OAI	Open Air Interface
CORE	Core Emulator
FPGA	Field-Programmable Gate Array
CPU	Central Processing Unit
GPU	Graphic Processing Unit
SoC	System on Chip
NOC	Network on Chip
FSM	Finite State Machine
EP	Event Pattern
UDG	Unit Disk Graph
QUDG	Quas-Unit Disk Graph
GPGPU	General-Purpose Processing on Graphic Processing Unit
CIE	Cloned Independent Events
IFE	Independent Foreign Events

Chapter 1

Introduction

1.1 Motivation and Objectives

Over the last decades, we have witnessed a great progress and an increasing need for discrete event simulation in a large range of scientific application. In particular, discrete event paradigm is useful for time-stepped simulation, which provides leaps in time and asynchronous updates (at potentially staggered virtual times) to different parts of the state.
The distributed DES was recognized as mostly adequate for large scale and complex simulation since it supports distributed features. In that context, network simulation was a leading research area that provides a significant portion of recent innovation in term of discrete event simulation theory and practice. Especially, large scale simulation appears as a real need when targeting the study of large systems evolving hundreds of thousands of simulated elements such as network nodes. Distributed simulation that relies on independent logical processes (LP) which communicate through message is enough for traditional large scale simulation where nodes are easily isolated. This paradigm requires minimizing the interaction between LPs compared to that within each LP to maintain the efficiency of the system. However, when targeting to simulate intensive interactive systems such that of modern wireless and mobile network, this paradigm becomes obsolete: we need either a very large LP that cover all dependents elements, or a low communication latency for inter-LP communication. Indeed, recent hardware provides several new features that increases the available computing power but requires a modified software architecture. The main hardware innovation is the usage of multi-core resources which enable parallel processing. Currently, the usage of multi-computing resource on the same LP remains in exploration phases. One fundamental corner of the usage of many computing resources is the parallel event scheduling that aims to execute several events concurrently while conserving the simulation correctness. Accordingly, the main research motivation of this thesis is the scalability issue of discrete event simulation (DES) in view of the heterogeneous computing. This thesis emphasis fours objectives:

- Studying and evaluating the pertinence of using emerging computing solution on large scale simulation.

- Reconsidering the event-driven simulation concept to maximize the usage of heterogeneous computing solutions.

- Designing a scalable methodology for general purpose, large scale and intensive simulation.
- Proposing a new scheduling approach that unlocks the DES bottleneck.

1.2 Contributions Storyline

In the 2010/2011 time frame, when we started to work in this thesis, we set as objective to increase the scalability of the intern simulation/emulation framework *OAI*. However, this goal required that we first transform the platform, from the concept of ***rapid validation tool*** to that of ***scientific experimentation tool***. Thus, before addressing the large scale issues, ensuring the reproducibility is a primary requirement that we must guarantee. Therefore, we design an experimentation methodology that defines a five-stepped workflow. That approach is largely inspired from existing methodologies used in large scale simulation and emulation frameworks [121, 117, 133]. When we started addressing the scalability issue, we first rely on a software engineering methodology that aims to identify computing bottlenecks and resolving them based on five possible approaches: GPU offloading, CPU parallelism, vector-processing parallelism, code optimization and compiling optimization. Each of these approaches provides a gain and induces an overhead. While this methodology delivers a real improvement in terms of performance and runtime, it becomes caducous when the overhead overpasses the gain. Accordingly, we admit the limit of this step once we reach the efficiency boundary.
With this experience in mind, we conclude that an architectural and fundamental work is required to propose an innovated answer to the scalability issue. Hence, we survey related works that discuss large scale simulation in the area of network simulation, and we determine four key issues:

1. Discrete event simulation is the most dominant approach to model large scale system while providing functional features such as the interoperability.
2. Event scheduling is a major issue in parallel simulation and more particularly, events-dependency detection is an expensive operation in terms of algorithm complexity.
3. The communication between distinct machines remains a major limiting factor of distributed simulation. Indeed, FLOPS becomes cheap while the communication is expensive.
4. Backward compatibility with sequential system is always maintained, transforming parallel simulation into an extension rather than a standard.

These observations give rise to a new principal of events managements which is *the generation of independent and parallel events, without subsequent control during the scheduling phase*. Even if it looks like as a promising idea, its realization into the OAI requires a major redesign work which may impact several projects that use

it as a validation and experimental platform. Thus, occurs the idea of creating a new simulator to validate the concept of generating grouped event in one hand and the pertinence of using the GPU as a simulation context for parallel execution on the other hand. We expect to increase the simulation efficiency by reducing the simulation runtime for a given scale, but also increasing the simulation scalability by allowing a unique logical process to handle much more simulated elements than what can be done by a CPU. The proposed simulation framework is named as Cunetsim: *CUDA network simulator*. Initial implementation provides six elementary services: mobility, connectivity, buffers management, broadcasting, traffic generator and power management. Based on a GPGPU software design, the simulation processing is defined by a list of time-stamped entries. At each entry, all nodes process the same event but on different data. Conceptually, the event-scheduler handles one entry for all nodes. The GPU driver and hardware transform that entry into a group of threads, each of which processes an event of one node. According to this methodology, we take advantage of the hardware management of threads and the automatic hardware synchronization of simulated nodes. In contrast with traditional parallel simulation, we avoid complex scheduling policies that aim to detect event dependency. We also compress the event representation which in turn reduces the scheduling cost. While earliest results were promising in term of efficiency and scalability, new requirements emerge: how to communicate with the outside world? How to handle complex scenarios with heterogeneous node definition? And more importantly how to overcome the GPU memory size limitation?
The interesting results of the initial Cunetsim implementation simplify a new framework to create a parallel and generic version that features parallel simulation over multiple GPUs. Further, we introduce a dynamic event scheduler that handles variable inputs and an extended event generator to support realistic scenario. The concept of an entry evolves from *being relative to all nodes* to *becoming relative to a sub group of simulated nodes*. Nevertheless, the simulation logic remains invariant: *an entry is a compressed representation of a group of events that will be executed in parallel*. The design pattern that we adopt for this solution is the Master-worker model where the master is a CPU process that manages the simulation and workers are assimilated to the GPU cores that execute events. Unfortunately, the event scheduling remains a centralized path where all entries must flow before being executed. Thus, when the number of execution resources increases (several GPUs, each of which includes thousands of cores), the event scheduler reappears as a bottleneck. Accordingly, we conclude this research phase with two results: first, it is possible to use a great computing power for a very large logical process efficiently using the grouped events concept. Second, the event scheduling remains the main bottleneck for large scale parallel simulation if it is centralized.
To further push the scalability boundary, we go one step back, and we reconsider the hardware architecture: *while the GPU is a powerful processor, it requires the CPU as a host. Moreover, the CPU is evolving to the multi-core architecture providing additional flexible computing resources*. Thus, we can combine both CPU and GPU on the same simulation rather than using the CPU solely as a manager. As from this

analysis the idea of the hybrid event scheduler emerged. The hybrid event scheduler is a pseudo-conservative scheduler that aims to execute events on the most adequate target between available executions resources. In particular, it switches grouped events to the GPU and isolated ones to the CPU. This approach maximizes the usage of heterogeneous computing resources on the shared memory. The gain from the proposed hybrid event scheduler proved to be significant and can reach up to 3x compared to traditional GPU scheduler.

The next intuitive step was to address the scalability further; we decide to distribute the simulation framework through a meta-computing infrastructure. Therefore, we incrementally design a three-tier software architecture donated as general purpose coordinator-master-worker (GP-CMW) model. It extends the *Master-Worker* model by introducing a top-level process which manages masters. In contrast with existing distributed simulation framework, the proposed architecture reduces significantly the management overhead through the network. The performance of a distributed simulation that relies on the GP-CMW architecture tends toward the maximal theoretical efficiency. The scalability of the system was validated on the largest European GPU-based supercomputer with 1024 LPs each of which simulates up to 1 million nodes. Accordingly, we had crossed a scalability boundary by combing parallel and distributed architecture into one global model that simplifies the management effort from the user point of view.

Finally, To validate the efficiency of the proposed concepts, we apply them to the popular network simulator NS-3 with the objective of a large scale deployment on the largest European hybrid super-computer(the TGCC Curie), which includes up to 288 GPUs and 1152 CPU. In particular, we customize the event scheduler of NS-3 and the event generation model in order to highlight the efficiency of the hybrid scheduling policy. This validation approach provides a proof of concept for the scalability and the robustness of the proposed model under production conditions. Further, we proved that maximizing the LP size until using the totality of a computing node is much more efficient than increasing the number of LPs within one computing node.

1.3 Thesis Structure

This thesis is structured around two parts. The first part includes background chapters that aim at providing a global view of the thesis context and investigation areas. In particular, the second chapter presents the network experimentation tools. The third chapter focus on the discrete event simulation and the fourth chapter highlights the hardware trends that lead the evolution of our work. The first part is concluded in the fifth chapter that presents related works which address large scale issues in network simulation or in DES. The second part of the thesis includes contributions chapters. The sixth chapter presents the Cunetsim framework that we use to explore the scalability horizons. The seventh chapter presents the hybrid event scheduler that aims at maximizing the hardware usage rate of het-

erogeneous hardware including CPU and GPU. The eighth chapter introduces the general purpose Coordinator-master-worker model that regroups the distributed and the parallel simulations on one optimized model. To conclude this part, the ninth chapter presents a study case of PADS methodology deployment through the $ns-3$ simulator. Finally, the thesis is concluded in the tenth chapter.

1.4 Publications

C1- **B.R. Bilel**,N.Navid, K. Raymond, B.Christian, *OpenAirInterface large-scale wireless emulation platform and methodology*, MSWIM'11, The 14th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, 31 october-4 Nov, Miami Beach,FL, USA.

P1- **B.R. Bilel**,N.Navid, *Cunetsim: a new simulation framework for large scale mobile networks*, SIMUTOOLS'12, Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques, 19-23 March, Desenzano, Italy.

C2- **B.R. Bilel**,N.Navid, *Cunetsim: A GPU based simulation testbed for large scale mobile networks*, ICCIT'12, Communications and Information Technology (ICCIT), 2012 International Conference on, 19-21 June, Hammamet, Tunisia.

C3- **B.R. Bilel**,N.Navid, M.S.M. Bouksiaa, *Hybrid cpu-gpu distributed framework for large scale mobile networks simulation*, DSRT'12, Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on, 25-27 October, Dublin, Ireland.

C4- **B.R. Bilel**,N.Navid, B.Christian, *Coordinator-Master-Worker Model For Efficient Large Scale Network Simulation*, SIMUTOOLS'13, Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques, 5-7 March, Cannes, France.

P2- **B.R. Bilel**,N.Navid,M.S.M. Bouksiaa, B.Christian, *Scalability demonstration of a Large Scale GPU-based Network simulator*, SIMUTOOLS'13, Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques, 5-7 March, Cannes, France.

C5- **B.R. Bilel**,N.Navid,M.S.M. Bouksiaa, B.Christian, *Hybrid scheduling for event-driven simulation over heterogeneous computers*, PADS'13, ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS), 19-22 May, Montreal, Canada.

P3- **B.R. Bilel**,N.Navid,M.S.M. Bouksiaa, B.Christian, *Events flow stability demonstration for a hybrid GPU-CPU scheduling in large scale network*, PADS Colloquium'13, ACM SIGSIM Conference on Principles of Advanced Discrete Simulation -PhD Colloquium-(PADS), 19-22 May, Montreal, Canada.

J1- **B.R. Bilel**,N.Navid, B.Christian,*General-Purpose Coordinator Master worker Model for Large Scale Distributed Simulation*, submitted to SMPT, simulation modelling practice and theory Journal.

1.5 Project Deliverables

D1- LOLA D4.1 *Specification of PHY&MAC Adaptations for the Target Architectures.*

D2- LOLA D5.1 *Testbeds Definition.*

D3- LOLA D5.2 *Testbed 2: First report on Integration of WP3 Traffic Models and WP4 L2 Algorithms on Testbed 1.*

D4- CONECT D4.1 *Testbed deployment and scenario specification.*

D5- CONECT D2.1 *: STATE-OF-THE-ART ON PACKET LEVEL COOPERATION TECHNIQUES.*

Part I

Background

Chapter 2

Network Experimentation

2.1 Introduction

Since the introduction of the first interconnected machines, the usage of a digital communication has grown rapidly and is part of all human and economy domain. Nowadays, networks are presents everywhere, in each home, company, car, and may be with each person on the earth. This evolution impacts both network size and data amounts that increase continuously. Accordingly, investigating or modifying any components of that gigantic system requires a complex validation process. Thus, after an initial theoretical analysis, algorithms and protocols are generally evaluated experimentally. To cope with the broad range of requirements that recent network experiments need to fulfill, different evaluation techniques have been established. In this chapter, we survey the most common experiment tools, and classify them according to their implementation granularity and realism level.

2.2 Network Experimentation Tools

A network experimentation tool is a framework that allows the experimenter to realize a given experiment. Several peripheral services can be added to this definition such as pre-definition service that help the experimenter to define the experiment or archiving/analyzing tools that simplify the usage of the output.
Regarding the literature reviews, classification works consider two fundamental parameters: the implementation granularity and the realism level. Accordingly, we highlight five principal approaches to network experimentation: real world field trial, real world testbed, emulation testbed , simulation testbed and hybrid testbed.

2.2.1 Real world Field Trial

Real world field experiment consists of deploying an experimental subject through real world infrastructure in order to study its behavior, efficiency or limits. In general, experimenters use real computers and network equipment, where they rapidly deploy their subjects of study (protocols, hardware, software). This approach was one of the first validation methods in network research and industry because its initialization requirements are limited and it provides realistic results. Nevertheless, given the number and complexity of the factors involved in the current networks, real world fields display three limitations:

- Precision: the real world experiment is based on a black-box approach where the system is evaluated as a whole. Thus, it is difficult to isolate a part of the system in order to analyze its impact, behavior or performance. Consequently, debugging and profiling activities are incompatible with such method.

- Reproducibility: a real world experiment is by definition, achieved in an open context where real traffic and real equipment cohabit with experimental one. As a consequence, it is hard to control environmental parameters that may influence the experiment. Thus, the reproducibility of such experiments is limited and remains an open issue.

- Scalability: the scalability of real world field is relatively limited due to three reasons:

    1. The deployment cost
    2. The management complexity
    3. The monitoring issues.

2.2.2 Real world Testbed

Real world testbed experiment consists on the deployment of the studied subject on a dedicated experimental infrastructure that imitates real world conditions [128]. By agreement, the testbed can be isolated or integrated onto a real and production context [71]. In contrast with experimentation, testbed increases the control by providing formal experimentation methodologies. Nevertheless, the reproducibility and the scalability of such an experiment remain challenging. In fact, the complexity of controlling all relevant parameters in a field test is precisely why so many users often turn to setting up testbeds where they can measure or control a large number of environmental parameters of interest and easily reproduce the same experiment over and over again. There are three principal tendencies that guide tested investigation: (1) Increasing the realism level (2) increasing the control and (3) increasing the scalability.

1. Increasing the realism level implies that the testbed grows as close as possible to target networks by incorporating the totality of involved components that may influence the experimentation[34, 116]. However, experimentation control and scalability remain limited due to inherent cost of such solutions.

2. Increasing the experimentation control implies the specification of environmental characteristics such as the transmission medium, medium access and usage, traffic patterns, power consumption, etc. That feature is particularly important for wireless network understanding that requires a full control of the medium[11, 115]. Indeed, the scalability of such approach remains a restriction.

3. Increasing the scalability implies the usage of additional resources. However, according to the high cost of such testbed, the leading approach is to federate foreign testbeds into a large infrastructure that can be shared between involved teams [112, 40]. In fact, federated real world testbeds such as[70] increases the tradeoff between the experimentation reproducibility and its realism level compared to field experimentation.

2.2.3 Emulation Testbed

In the literature reviews, researchers talk about emulation when a sub part of the studied system is partially modeled using a mathematic or software representation while the other parts of the system are real. Another work considers an experiment as an emulation when the environment of the studied subject becomes fully controlled. Regarding network emulation frameworks, literature reviews highlight the control of the experiment. The objective is to be able to configure all the parameters of an experiment execution. Depending on the framework goal, modeled components may be either the medium, the hardware or software. Emulating the network medium is typically used in wireless context where the channel specification and modeling are under improvement [50, 29, 63].
In particular, FPGA based framework such as [66, 19] provide a full control on the medium environment, propagation, collision and dispersion. Using FPGAs as development context is also used in emulation context to study the radio medium [18]. Modeling the hardware is commonly used in network emulator that addresses protocols development such as CORE [6]. The main goal of such approach is to execute real protocol implementation that can be used in testbeds and field experimentations. Modeling the software is mainly used when the protocol stack and/or a part of the software is not the subject of study such that its real implementation did not improve the experimentation but slows the execution. A good example of software emulation is trace based network emulation [93]. Several works that can be classified as an emulation framework or approach, are positioned between [24, 59, 52, 79, 89]. Indeed, the scalability of emulation frameworks remains restricted due to the imposing cost per emulated node whatever the used emulation approach.

2.2.4 Simulation Testbed

In the literature reviews, researchers talk about simulation when the experiment consists of using models to represent the studied system. In fact, each component of the simulated system is represented using a model. External elements such as the environment, foreign systems are also represented using models. Models can be defined using more or less complex algorithms [22, 68].
The challenging part of a network simulation is the representative implementation of the real system in a model [65]. To cope with the original system's complexity, the simulation model requires abstractions and simplifications, which in best-case do not produce an impact on the definitive result. Moreover, the run-time of the

simulation depends on the degree of detail in the implementation [148]. Hence, the network simulation is an appropriate tool to evaluate specific effects, but due to the modeling approach, they may not capture correctly the entire system behavior. Thus, when interpreting the results of a given simulation, it is perpetually required to take into account these limitations.
There are two main categories of simulations: continuous [25] and discrete [130] simulations. In continuous simulations, the model is represented as a set of differential equations. They are often used for researching physical phenomena, such as aerodynamics or hydraulics, which require numerical solutions. On the other side, discrete simulations implement the model as a sequence of events at discrete time points. In computer networking, the DES is an important instrument for protocol development and evaluation.
The major advantage of network simulation is unquestionably its scalability. However, its realism level is directly related to the correctness of the used models. Recent network simulators such as $omnet++$ [137] and $NS-3$ [60] provide a good trade-off between models simplicity and quality. Production network simulators such as *opnet* [56] pushes the balance further by incorporating the real world code such as the IP stack implementation, until grazing the definition of network emulator rather that of simulator. In particular we highlight the ability of several simulation testbeds such as $NS-3$ to execute external software source code that can be exactly which will be used on real implementation. The feature is denoted as *direct code execution* [75].

2.2.5 Hybrid Testbed

Each of the previous network experimentation tools presents strengths and weaknesses in terms of scalability and realism level. These features seem conflicting since increasing the realism requires complex model or real implementation while increasing the scalability requires a reduced complexity per simulated element. However, the need of realistic and large scale experiment rises proportionally to the growth of networks scale. Moreover, experimentation that involves heterogeneous components such as wireless and wired nodes gain momentum and credibility. Accordingly, the idea of combining the usage of different experimentation tools on the same framework appears the most adequate to cope with both requirements. In the literature reviews, we distinguish two main categories of hybrid experimentation tools: the federating tools of the same class and the federating different experimentation class tools.
Federated experimentation frameworks have a directed line that defines a global goal such as *considering very large Internet experimentation* or *achieving a very large scale packet level simulation* [136, 87]. For example, the Planetlab project federates different type of experimental testbeds [113, 51]. In this context, we note that the HLA protocol is a standardization that allows the federation of heterogeneous simulators[35]. While that category of hybrid experimentation targets to maximize the scalability of a given tool/framework by increasing its distribution level, it is still

limited by the inherent characteristics of the considered tool (testbed, emulation or simulation).
In contrast, federating different experimentation tools provides a sophisticated and flexible combination between local realistic and global large scale experimentation [53]. For example, when studying the integration of 4G equipment on the Internet, a typical hybrid experimentation will emulate 4G nodes using a full protocols stack implementation on radio devices using OpenAirInterface, emulates Wi-Fi and 3G nodes that share the same medium using CORE and simulates the network backbone using NS3. In that sense, the NEPI project [77] is one of the federating framework that aims to provide a global methodology and interface to combine a variety of network experimentation tools. To face the increasing complexity of such experimentation, federated frameworks must synchronize the global time of the experiment and primary components definition. Generally, the referential time of hybrid network experimentation is a real time while the primary components is always the node.
Finally, it has to be mentioned that tendencies of sophisticated experimentation tools such as NS3 and CORE is to provide dual or triple capabilities including emulation, simulation and visualization, which appears as an emerging feature of such tools [149].

2.3 Network Experimentation Aspects

In previous sections, we classify five types of network experimentation tools, based on their realism level. Realism level and scalability can be used to classify experimentation tools. Table 2.1 summarizes a realism-based classification approach that focus on most common examples of each category. Nevertheless, the realism level remains one aspect of the experiment. A synthesis of the literature reviews highlights five major axes that define a given experimentation tool.

1. *The architecture*: the tool architecture summarizes the concepts that define the achievement of the tool. It includes the experiment base which defines how the experiment is described in the tool (components definition, topology, traffic description). The second aspect of the architecture is the component base. In fact, each component can be defined using realistic/simplified definition. Finally, the simplification aspect considers the user point of view. It is relative to the simplicity of the usage of the tool and not to the quality of the experiment.

2. *Scientific experiment features*: in order to be considered as a scientific experiment, the experimentation tool must address three features: the reproducibility, the revisability and the traffic quality. The reproducibility is the ability to reproduce the same experiment, expects to produce the same output and observes the same behavior. The revisability concerns the capacity of the experimenter to explore the studied system during the experimentation in order

to extract meaningful results. The granularity of the exploration is important in the revisability of a given experiment (including dynamic debug and profiling). The traffic quality is an experimentation feature that may define the scientific quality. Thus, the ability of a given testbed to play real traces or to generate realistic traffic is an important feature.

3. *The effectiveness*: the effectiveness of an experimentation tool defines its capacity to achieve the experiment rapidly without interference between the experiment and the experiment management. The effectiveness of a given tool also depends of its scalability, defined as its capacity to scale in terms of size of the simulated scenario (number of element, duration of the experiment, traffic load, and the scope of the space). In table 2.1, we characterize the effectiveness aspect by three parameters: the scalability, the overhead and the efficiency. The first defines the ability to scale, the second is the tradeoff between the simulation and the management processing. Finally, the efficiency of a given experimentation tool reflects its processing runtime.

4. *The cost*: the economic aspect is an important factor that determines the notoriety and the usage of a given tool. This aspect includes three characteristics: the initialization cost, the operating cost and the maintenance cost. The first defines the cost of the experiment initialization including hardware and software. The second defines the cost of the usage during an experiment (i.e. the power). The third aspect is relative to usage in the long term and describes the evolution capacity of the tool.

5. *The usability*: In contrast with previous aspects, the usability is relative to the usage experience. It includes the methodology of the experiment (that also impact the scientific aspect), the simplicity of the deployment which is very critical when addressing distributed simulation and the experiment setup that implements the experimentation methodology.

In table 2.1, we highlight the correspondence between the five classes of tools and the experimentation aspects.

Table 2.1: Characteristics of exciting experimentation tools

Characteristic	Filed test	Testbed	Emulation	Simulation	Hybrid
Experiment bases	++ Real	+ Real	-Real/model	Model	+/-Real
Components bases	++Real	++Real	+Real	Algorithm	Variable
Simplifications	absent	Possible	Definable	Abstraction	Variable
Reproducibility	-Insight	Insight	+/-Accurate	++Accurate	+Accurate
Revisability	+/-Insight	+Insight	+Accurate	+Insight	++Accurate
Traffic	++Real	++Real	+/-Real	+/-Real	+Real
Scalability	−−	-	++	+/-	
Overhead	− − −	−−	-	+/-	+/-
Efficiency	− − −	−−	-	++	+/-
Initialization Cost	− − −(expensive)	−−	+/-	++	+
Operating Cost	−−	−−	−−	++	+/-
Maintenance Cost	− − −	−−	-	++	+
Methodology	− − −	++	++	+++	+++
Deployment	++	+	+/-	−−	+/-
Setup	− − −	-	+/-	++	+

Chapter 3

Discrete Event Simulation

3.1 Introduction

Discrete event simulation (DES) is the process of modeling the operation of a given system as a discrete sequence of events in time. Each event occurs at a given time-point of the simulation time axis and marks a change of state in the system [120]. Discrete-event simulation is largely used as a major software design approach in scientific simulation. The reason behind the success of discrete-event based simulation in computer networking is that the simulation paradigm fits very well to the considered systems. In fact, DES provides a simple yet flexible way to achieve complex experiments and to study the behavior of the systems under different conditions. In the remainder of this chapter, we provide a brief introduction to discrete-event simulation in Section 3.2 and we summarize the principals of parallel discrete event simulation (PDES) in section 3.2 . In particular, we highlight issues and principles of PDES.

3.2 Discrete-Event Simulation

In this section, we first introduce the basic terminology used in literature and describe their relationship. Second, we present the principal of DES and we survey the concept of time driven event scheduling. For an in-depth introduction the reader is referred to [80, 142].

3.2.1 Terminology and Components

There are three notions shared among the majority of DES: the entity, the system and the discrete system.

- An *entity* is an abstraction of a particular subject of interest. An entity is described by its attributes, e.g., an entity packet could have attributes length, source and destination addresses. The term *object* is often used as synonymous.
- A *system* is defined by a set of entities and their relationship. The set of entities and their relationships fulfill a certain purpose, i.e., the system has a certain goal that it tries to achieve.
- A *discrete system* is a system whose state, defined by the state of all entities of the system, changes only at discrete points in time. The change of the state

is triggered by the occurrence of an event. What an event exactly is, depends mainly on the system and on the goal of the study., e.g. sending and receiving a packet, moving in the space or updating the battery state.

Usually, the system of interest is quite complex. In order to evaluate its performance by means of computer simulation a **model** is built. The model is a *software representation* of the system, hence it consists of selected entities of the system of interest and selected relationships between the entities. By agreement, the model is a system itself. In computer simulations, it is always the model that is considered, mainly to reduce the involved complexity and the associated cost and effort.

3.2.2 The Principle

The idea of a discrete-event simulator is to jump from one event to the next, whereby the occurrence of an event may trigger changes in the system state as well as the generation of future events. The events are recorded as event descriptor (also known as the event notice) in the future event list (FEL), which is an appropriate data structure offen a time ordered structure to manage all the events in the discrete-event simulation. An event descriptor is composed of at least two information (time, type) where time specifies the time when the event will occur, and types provide the kind of an event. The future event list should implement efficient functions to insert, to find, and to remove event descriptor, which are placed in the future event list. With every discrete event time *ti* a snapshot of the system is created (stored in memory) that contains all required data to progress the simulation. In general, all discrete-event simulators share the following components:

- *System state*: a set of variables that describe the state of the system.
- *Clock*: the clock gives the current time during the simulation.
- *Future event list*: a data structure appropriate to manage the events
- *Statistical counters*: a set of variables that contain statistical information about the performance of the system.
- *Initialization routine*: a routine that initializes the simulation model and sets the clock to 0.
- *Timing routine*: a routine that retrieves the next event from the future event list and advances the clock to the occurrence time of the event.
- *Event routine*: a routine that is called when a particular event occurs during the simulation.

3.3 Parallel Discrete Event Simulation

Ever since discrete event simulation has been adopted by a large research community, simulation developers have attempted to draw benefits from executing a simulation

on multiple processing units in parallel. Hence, a wide range of research has been conducted on Parallel Discrete Event Simulation (PDES). In the remainder of this section, we survey the challenges of parallel DES, common architecture and major synchronizations algorithms.

3.3.1 Discrete Event Simulation Limits

Technological systems are becoming increasingly complex with the emergence of new technologies that combine wide interconnectivity with composite structures and architectures. In general, two orthogonal trends in terms of complexity can be identified: (i) an increase in structural complexity and (ii) an increase in computational complexity. Both impose high demands on the simulation architecture and the hardware executing the simulations.

We denote the size of a simulated system as an indicator of the structural complexity of a simulation model. In what concerns network research, growing systems like peer-to-peer networks and ubiquitous mobile networks, induced an enormous increase in the size of communication systems. Such large systems typically add complex behavioral characteristics which cannot be observed in networks of smaller size (e.g., testbeds) or captured by analytical models. Thus, in order to study those characteristics, simulation models a large numbers of simulated network nodes. Since every network node is represented in memory and triggers events in the simulation model, memory consumption and computation time increase significantly. Even if the investigated network is relatively small, computational complexity becomes an important factor if the simulation model is highly detailed and involves extensive calculations [47]. In particular wireless networks which make use of advanced radio technologies such as OFDM(A) [72] and Turbo Codes [14] fall in this category. Sophisticated radio propagation models, interference modeling, and signal coding models increases the overall complexity.

Simulation frameworks aim to compensate these issues by enabling simulations to be executed in parallel on multiple processing units. By combining memory and computation resources of multiple processing units, simulation time can be reduced at the cost of higher memory requirements and management overhead. Although this approach is known for more than two decades [42, 44, 110], recent technological advances considerably reduce the hardware cost of parallel computing infrastructure. Thus making such hardware available to a large research community which hand over the spotlight on discipline [86, 127].

3.3.2 Principles of Parallel Discrete Event Simulation

The approach taken by PDES is to divide a simulation model in multiple instances which are executed on independent processing units in parallel. The central challenge of PDES is thereby to maintain the correctness of the simulation. Any simulation framework exhibits three central data structures: i) state variables of the simulation model, ii) a timestamped list of events, and iii) a global clock. During

a simulation run, the scheduler continuously removes the event with the smallest timestamp (*emin*) from the future event list(FEL) and executes the associated handler function. T denotes the timestamp function which assigns a time value to each event and E is the set of all events in the event list. While the handler function is running, events may be added to or removed from the event list. Choosing *emin* is crucial as otherwise the handler function of an event *ex* with $T(emin) < T(ex)$ could change state variables which are later accessed when *emin* is handled. In this case the future (*ex*) would have changed the past (*emin*) which we call a causal violation. Thus, we formulate the central challenge of PDES as follows: Given two events e1 and e2, decide if both events do not interfere, hence allowing a concurrent execution, or not, hence requiring a sequential execution. Parallel simulation frameworks employ a wide variety of synchronization algorithms to decide this question. The next section presents a selection of fundamental algorithms and discusses their properties.

3.3.3 Parallel Simulation Model and Algorithms

A parallel simulation model is composed of a finite number of partitions which are created in accordance to a specific partitioning scheme. Three exemplary partitioning schemes are i) space parallel partitioning scheme, ii) channel parallel partitioning, and iii) time parallel partitioning. The space parallel partitioning scheme divides the simulation model along the connections between simulated nodes. Hence, the resulting partitions constitute clusters of nodes. The channel parallel partitioning scheme bases on the assumption that transmissions that utilize different (radio) channels and/or mediums do not interfere with each other. Thus, events on non-interfering nodes are considered independent. As a result, the simulation model is decomposed in groups of non-interfering nodes. However, channel parallel partitioning is not generally applicable to every simulation model, thus leaving it for specialized simulation scenarios. Finally, time parallel partitioning schemes subdivide the simulation time of a simulation run in time-intervals of equal size. The simulation of each interval is considered independent from the others under the premise that the state of the simulation model is known at the beginning of each interval. However, the state of a network simulation usually comprises a significant complexity and is not known in advance.

3.3.3.1 Local Causality Constraint

A discrete-event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages obey the local causality constraint if and only if each LP processes events in non-decreasing time stamp order. In practice, the number of LPs (i.e., partitions) is equal to the number of CPUs provided by the simulation hardware. Consequently LPs directly map to physical processes. Furthermore, the timestamped and message-based communication scheme constitutes two important properties. First, they allow a transparent execution of LPs

either locally on a multi-CPU computer or distributed on a cluster of independent computers. Second, and more importantly, timestamps provide the fundamental information used by synchronization algorithms to decide which events to execute and to detect causal violations. In the literature, there are two classes of synchronization algorithms: conservative and optimistic algorithms. While conservative algorithms aim to strictly avoid any causal violation at the time of the simulation run, optimistic algorithms allow causal violations to occur, but provide means for recovering.

3.3.3.2 Conservative Synchronization Algorithms

Conservative synchronization algorithms strive to strictly avoid causal violations during a simulation run. Hence, their central task is to determine the set of events which are safe for execution. In order to decide on this question, conservative algorithms rely on a set of simulation properties [43]. The Lookahead of a LP is the difference between the current simulation time and the timestamp of the earliest event it will cause at any other LP. The Earliest Input Time (EIT) denotes the smallest timestamp of all messages that will arrive at a given LP via any channel in the future. Accordingly, the Earliest Output Time (EOT) denotes the smallest timestamp of all messages that a given LP will send in the future to any other LP. Based on these definitions, a LP can safely execute all events which have a smaller timestamp than its current EIT since it is guaranteed that no messages with a smaller timestamp will arrive later.

Null-Message Algorithm

The simple approach presented above does not solve the synchronization problem entirely as it can cause the simulation to deadlock. This problem is addressed by the Null-Message Algorithm (NMA) which was first introduced by Misra and Chandra [317]. The algorithm uses nullmessages, i.e., messages which do not contain simulation model related information, to continuously increase the EIT of all neighboring LPs. For this purpose, null-messages carry the LP's current EOT timestamp, which is determined by adding the lookahead to its current local time. Hence, nullmessages can be considered as a promise of an LP not to send any message with a smaller timestamp than EOT in the future. Upon receiving a null-message, each LP updates its EIT to a potentially greater value. If the updated EIT has advanced beyond events in the event queue, those are now considered safe for execution. This algorithm guarantees to prevent deadlocks if the simulation model does not contain zero-lookahead cycles. However, this algorithm implies a significant management overhead. In fact, each synchronization algorithm imposes two types of overhead: messaging overhead which is caused by sending (simulation model) messages to LPs on remote machines and the actual synchronization overhead which is caused by blocking and additional (synchronization) messages that are only used by the algorithm (e.g., null-messages). While messaging overhead is a property of the simulation model, the synchronization overhead is a property of a particular algorithm.

3.3.3.3 Optimistic Synchronization Algorithms

In contrast to conservative algorithms, optimistic synchronization algorithms allow LPs to simply execute all events (in time stamp order) as they come in, but without ensuring that causal violations will not occur. This probably counter-intuitive behavior is motivated by the observation that conservative algorithms sometimes block LPs unnecessarily: Often not enough information is available to mark a certain event safe, although it actually is. Hence, the simulation performance is reduced significantly. Thus, optimistic algorithms assume that an event will not cause a causal violation. This approach has two primary advantages: First, it allows exploiting a higher degree of parallelism of a simulation model. If a simulation model contains two mostly independent partitions, which interact only seldom, only infrequent synchronization is actually needed. Second, the overall performance of the parallel simulation depends less on the lookahead. Thus making it attractive to models with small lookahead such as in wireless networks. Clearly, the downside is that a causal violation leaves the simulation in an incorrect state. As a result, optimistic algorithms provide recovery mechanisms: during a simulation run, the PDES engine continuously stores the simulation state. Upon a causal violation, the simulation is rolled-back to the last state known to be correct.

Chapter 4

Hardware Trends

4.1 Introduction

This chapter aims to highlight the hardware evolution that has accrued in the last decades. In particular, we survey parallelism and optimization features that concern CPUs and the emergence of dedicated co-processor that democratizes the large scale parallel computing. In this context, we present the GPU as a massively multi-core chip and the computing accelerator as a dedicated multi-core device. In the remainder of this chapter, we first survey CPU architectures and features. Secondly, we present an overview of current GPU state of the art. Thirdly, we summarize the concept of multi-core accelerator that was inaugurated by the the Xeon Phi processor. Finally, we conclude the chapter by a survey of parallel programming APIs for such hardware.

4.2 Evolution of Computing Chips

4.2.1 CPU: Historical Evolution and Trends

Historically, the CPU was designed to achieve all computing tasks requested by the OS. Since the introduction of the 8086 micro-processor and the democratization of the PC, the CPU improvements tell two roads: increasing the CPU frequency and extending the instructions set.
Nevertheless, increasing the frequency of the computing chip induces a phenomenal increase in thermal settings and power consumption. Accordingly, in the beginning of the previous decade, CPU manufactures recognize that increasing the frequency cannot be the future development approach. The common trend is to fit multiple CPU cores within the same chip. First attempts was really the juxtaposing of two independent CPUs which in turn limits any advanced collaboration and saturates the internal bus. Progressively, CPU architecture evolves and becomes composed of several computing cores and on-chip memory stages. The communication between different cores is progressively optimized, and the management of the distributed memory becomes mature. However, the concept of multi-core CPU relies on the independence of each core such that each of them is able to execute any task. Consequently, all cores are built following the same complex architecture which in turn reduces the ability to maximize the number of integrated cores.
On the other hand, the enhancement of the instructions set remains relevant. In particular, the introduction of vector instructions was pertinent for scientific and

multimedia usages. These instructions allow the processing of several words in one clock cycle. In the current state of the art, it is possible to process up to 4 words (256 bits) in parallel by each core using the AVX instruction set. AVX2 promises the processing of 8 words in the next CPU generation.
In parallel, CPU manufacturers continue to improve CPU architecture to provide more and more features. The trend is that, between two consecutive generations, the gain is between 10% and 20%. An additional relevant feature is the simultaneous multithreading (SMT), denoted as the hyper-threading technology by Intel and partially implemented by AMD. The concept relies on an advanced instruction scheduling that allows each hardware core to emulate the execution of two (or more) threads. Each of which is assumed to be executed on one virtual core. Depending on the optimization of the software, the relative gain could vary (between 10% and 25%).

4.2.2 GPU: Historical Evolution and Trends

By definition, a graphics processing unit (GPU) is a co-processor that ensures the graphical rendering. The evolution of the current graphics processor begins with the introduction of the first 3D devices. The early graphics systems featured a fixed function pipeline (FFP), and architecture following a very rigid processing path utilizing almost as many graphics APIs as there were 3D manufacturers. While the 90s period counted many innovative GPU manufacturers, the early of 2000s was marked by the concentration of the GPU industry around two actors: NVIDIA and ATI. The main evolution of that time was the introduction of more and more flexible programming API such as the DX7 and the OpenGL (1999-2000) and the support of 32-bit floating point computing with GPU. A revolution in the GPU industry was the beginning of the first unified graphics and computing GPU, programmed in C with CUDA. The Geforce 8800 was the first CUDA compliant GPU that includes up to 128 CUDA cores while the most powerful CPU of that time includes two cores. The GPU development grew in two directions: simplification of the programming interface and the increase of the number of computing cores. Thus, the OpenCL API is mostly supported by the majority of chip manufacturers, including AMD, NVIDIA, Intel and ARM.
It is relevant to highlight that current GPUs are throughput-oriented devices made up of hundreds of processing cores. They maintain a high throughput and cache memory latency by multithreading between thousands of threads. GPUs rely on a two level hierarchical architecture. It is made of vector processors at the top level, termed streaming multiprocessors (SMs) for NVIDIA GPUs and SIMD cores for AMD GPUs. Each vector processor contains an array of processing cores, termed scalar processors (SPs) for NVIDIA GPUs and stream processing unit for AMD GPUs. All processing cores inside one vector processor can communicate through an on-chip user-managed memory, termed shared memory for NVIDIA GPUs and local memory for AMD GPUs. The CUDA [147] and OpenCL [132] APIs share the same SPMD (Single Program Multiple Data) programming model. CUDA vir-

tualizes SMs as blocks (equivalent to workgroups in OpenCL) and SPs as threads (equivalent to workitems in OpenCL), which allow programmers to execute thousands of threads and blocks across different generations of GPUs regardless of the amount of physical processors. A key concept of the CUDA programming model is the warp, equivalent to the wavefront in AMD GPUs. A warp is a group of 32 threads that execute in lockstep in a SIMD fashion. Because the GPU architecture shares a single instruction unit for all threads in a warp, a warp is the smallest unit of work.

4.2.3 Emerging Solutions

In addition to the two dominant chips on the computing area, (CPU and GPUs) the last few years have marked the emergence of new categories that aim to provide innovative solutions for existing limitations. In particular, system on chip (SoC) and CPU accelerators, have a good potential as a computing support for future large scale system.

4.2.3.1 System On Chip

A system on a chip or system on chip (SoC) is an integrated circuit (IC) that integrates all components of a computer or other electronic system into a single chip [49]. It may contain digital, analog, mixed-signal, and often radio-frequency functions-all on a single chip substrate. In what concerns this study, we focus on SoC that integrates CPU and GPU cores in the same SoC that integrates heterogeneous Cores.
Recently AMD (Fusion APUs) [20], Intel (Sandy Bridge) [55, 150] and ARM (MALI) [103] proposed new solutions that integrate general purpose programmable GPUs together with CPUs on the same die. In this computing model, the CPU and GPU share memory and a common address space. These solutions are programmable using OpenCL [132] or solutions such as DirectCompute [151, 9]. Integrating a CPU and GPU on the same chip has several advantages. First it is cheaper because of system integration and the usage of shared structures. Secondly, this promises to improve performance because there are no data transfers between the CPU and GPU. Thirdly, programming becomes simpler because explicit GPU memory management is not required. Not only does CPU-GPU chip integration offer performance benefits but it also enables new directions in system development.

4.2.4 Multi-Core accelerator

The concept of multi-core accelerator is a new approach, mainly boosted by Intel. Thus, the MIC architecture was prototyped and revealed for exclusive partners before proposing the XEON Phi coprocessor to the community. The product is packaged on a PCIE device that regroups a multi-core CPU (57-61 cores) and a dedicated memory. While that accelerator is underclocked in comparison with the

main CPU, it has the advantage of being x86 compliant and easily used via traditional compilation tools. In contrast with the GPU, the accelerator emphasizes three features: first it provides fully independent cores which may execute independent threads. Secondly, it is compliant with existing x86 code. Thus, the redesign requirements are natively limited. Thirdly, the computing power in single and double precision remain competitive compared to that of GPUs. In addition, Intel highlights the importance of combing powerful core for isolated and non-parallelizable tasks with the usage of many medium cores for parallelizable tasks and processes.

4.3 Parallel Programming: Models and API

The purpose of parallel computing is to raise the performance by executing the application on multiple processors. Even if parallel computing is traditionally correlated with the HPC community, it is becoming more common for mainstream computing due to the recent emergence of multi-core architecture. However, the optimal usage of such hardware requires adequate programming tools that maximize the parallelism features. In that sense, there are several APIs which provide parallel and distributed programming support. In particular, we identify five representative APIs: the Pthreads, the OpenMP, the MPI, the OpenCL and the CUDA.

4.3.1 Pthreads

Pthreads is the acronym of Portable Operating System Interface (POSIX) Threads. Pthreads is implemented as a header (pthread.h) and a library for creating and handling each of the workers called threads. Worker administration in Pthreads requires to explicitly manage the threads lifecycle from the creation to the exit. Furthermore, the definition of workload division and mapping must be explicitly defined by the software designer. To preserve critical section, i.e. the part of code that accesses shared data, Pthreads grants mutex (mutual exclusion) and semaphore. Mutex authorizes only one thread to access a critical section at a given time, whereas semaphore permits different threads to access a critical section.

4.3.2 OpenMP

OpenMP is a generic specification that considers shared memory parallelism. It consists of a collection of runtime library routines and environment variables that simplify the parallel programming in a shared memory context. OpenMP is available as an extension of Fortran, C and C++ programs. The primary worker of the OpenMP design is threads. The worker management is implicit and relies on the usage of pre-compilation Pragma directives which indicate that a given section could be executed in parallel. The number of parallel threads is an environmental variable that depends on the hardware capacities. Thus, unlike Pthread, the developer involvement is limited to the design of threads interaction and data usage.

Workload partitioning and task-to-worker mapping require a relatively little programming effort. OpenMP also abstracts the workload distribution among workers and how tasks are to threads.

4.3.3 MPI

Message Passing Interface (MPI) is a specification for message passing operations. The concept of MPI considers that each worker is an independent process. MPI is currently the standard for developing HPC applications on distributed memory architecture. It provides language bindings for C, C++, and Fortran. Some of the most popular MPI implementations include OpenMPI, MVAPICH, MPICH, GridMPI, and LAM/MPI. The control of the workers is ensured using a command-line tool (or using scripts) and the system ensures the management of different processes among available hardware according to the given configuration. Programmers have to control what tasks are to be computed by each process. Communication among processes adopts the message passing paradigm. MPI broadly classifies its message-passing operations as point-to-point and collective. MPI Barrier is used to specify that synchronization is needed. The barrier operation blocks all processes until being reached by all processes that participate on the barrier.

4.3.4 CUDA

The Compute Unified Device Architecture (CUDA) is a general purpose scalable parallel programming model for writing highly parallel applications. It provides several key abstractions: a hierarchy of thread blocks, shared memory, and barrier synchronization. In the current state of the art, CUDA is exclusively compliant with Nvidia GPU. The model views a parallel system as a couple of a host (i.e. CPU) and devices (i.e. GPU). Parallel computing tasks are done in GPU by a set of parallel threads. The model organizes the threads on a two-level hierarchy, namely block and grid. Block is a set of cloned threads, each of them is identified by a Tid (thread id) while the grid is a set of loosely coupled of blocks. The management of the workers is implicitly achieved by the CUDA driver. Thus, programmers specify the parameters of the grid and block required to process the given work. It has to be mentioned that thread synchronization is done implicitly using available routines such as the function syncthreads().

4.3.5 OpenCL

OpenCL is a new standard for job-parallel and data-parallel heterogeneous computing on a range of modern CPUs, GPUs, DSPs, and other microprocessor designs. OpenCL provides abstractions and a set of programming APIs based on past successes with CUDA, TBB, and other programming toolkits. OpenCL defines a set of core functionality that is supported by all devices, as well as optional functionality that may only be implemented on high-function devices, and includes an extension

mechanism that allows vendors to expose specific hardware features and experimental programming interfaces for the benefit of software developers. Although OpenCL cannot mask significant differences in hardware architectures, it does guarantee portability and correctness. This makes it much easier for a developer to begin with a correctly functioning OpenCL program tuned for one architecture, and create a correctly functioning program optimized for another architecture.

Chapter 5

Related Work

5.1 Introduction

In this chapter, we present the related work that address the issues of parallel and distributed simulation over heterogeneous computing nodes. The chapter is composed of three sections. The first section introduces the most common approaches which address large scale simulation. The second section highlights the event scheduling issues and the third section summarizes most common optimizations that emerge in the distributed simulation domain.

5.2 Large Scale Simulation

In the literature reviews, there are three major approaches to deal with large scale simulation: (1) *CPU-based parallel and distributed simulation*, (2) *partial acceleration using specific Co-processor* and (3) *the fully GPU approach*.

1. In a *CPU-based parallel and distributed simulation* [97], the platform is composed of several simulation instances that collaborate to ensure the simulation in a given way. The most common distributed and parallelization of CPU-based simulation are:

   - Spatial distribution: each instance simulates a part of the global space and/or population.
   - Functional distribution: each instance ensures one(s) task/ function(s) for the global simulation.
   - Collaborative distribution: includes dynamic distributed simulations and the simulation which combines the usage of several CPU cores per simulation instances. In most common terminology, we talk about a federated approach to summarize this category.

   Such a federated approach makes use of existing models and provides a rapid parallelization of existing sequential simulators [107]. In this context, GTNetS is an experimental framework that democratizes the usage of flat/hierarchical architecture to manage large scale network simulation [118]. Afterwards, the development experience of GTNetS was applied to the open source network simulator $ns-3$ with good performance in term of scalability [69]. However, such approach introduces a significant overhead due to the synchronization

among different processes and/or machines and requires sophisticated and expensive simulation infrastructure [96]. This overhead may drastically increase in a mobile environment if the network topology and machine mapping is not dynamically managed (e.g. through node migration). For the majority of CPU-based simulators, the performance degradation happens when the simulation combines the limiting factors such as mobility rate, number of nodes, and traffic load increases. Regarding distributed simulators, such performance degradation happens when the inter-machines communication increases. A scalability demonstration, based on the distributed NS-3 has carefully avoided the problem of the interaction between nodes in different simulation machines [69]. Even if parallel and distributed simulators have crossed a scalability boundary, they introduce new problems such as the cost of a simulated node, the strategy of initial nodes distribution and their migration across different machines.

2. The *partial acceleration* aims to increase the efficiency of the simulation locally by offloading the most CPU-intensive part of a given simulation from the CPU to a dedicated co-processor. The FPGA was widely used as an acceleration solution[30] however, in some recent approaches, the GPU is used to offload intensive computing tasks such as channel modeling [10] and queuing [101] within the simulator. Recent studies recommend the GPU usage for more general-purpose simulation [109], or even as a GPU-accelerated simulation architecture when accuracy and runtime performance are both critical [7]. Thus, the GPU becomes an increasingly attractive alternative to the expensive CPU-based parallelism, with significant computational power at a relatively low cost. With the advent of the *GeForce*8 series GPU in 2006 and the compute unified device architecture (CUDA) [94], the GPU becomes an available computing support. Even if this approach reduces significantly the computing time, the simulation remains primarily in the CPU which continues to be the system bottleneck in large scale scenarios. Further, a continuous data transfer between the GPU memory and the CPU one presents a serious limitation.

3. *The fully GPU* approach aims to offload the totality of the simulation tasks on the GPU space. In addition, the required memory will be exclusively on the GRAM. This approach offers three advantages:

   - The impact of the CPU is minimized, in contrast with traditional parallel simulation.
   - The memory transfer between the main memory and the GRAM is almost null during the simulation.
   - The synchronization latency between different simulation cores is reduced.

   However, the GPU is not fully X86 compliant, does not support CPU features, needs a particular software architecture to disclose its power and does

not support memory lock mechanism. Because of these constraints, the fully GPU simulation approach is poorly studied. In the same context, we propose a proof of concept, denoted as Cunetsim that uses the GPU as a main simulation environment and the CPU as a controller. Cunetsim is an experimental simulation platform allowing validation and experimentation of novel approaches. As opposed to previous works, Cunetsim is designed to provide an independent parallel execution environment for each simulated node. Nodes communicate through message passing based on the buffer exchange. Thus, the framework avoids the usage of global knowledge and increases the parallelism level. It is based on the master/worker model for a CPU-GPU co-simulation and provides hybrid synchronization model which maximizes the efficiency and guarantees the correctness of the simulation. The simulation exploits the large number of computing cores of the GPU to execute nodes in parallel and the high speed memory access to reduce node communication latency.

5.3 PDES Issues

Improving the efficiency and scalability of DES remains a challenging issue for modern modeling approaches that require complex and sophisticated representation. In such context, PDES is commonly used as a scalable and efficient solution when compared with sequential approaches [33]. PDES relies on the partitioning of the model over several logical processes (LP)s collaborating with each other to perform the whole simulation [111, 62]. However, respecting the simulation correctness while dealing with parallel execution makes event management extremely expensive. This is also acknowledged as one of the critical limitations of large parallel simulations, especially when dealing with heterogeneous resources, and raises two issues: data representation and event scheduling [110].

5.3.1 Data Representation

To store future events, most of the PDES frameworks use a sorted data structure. In the literature reviews, the efficiency of central data structures was largely studied for both sequential and parallel execution, e.g. central event list (CEL) [122]. Nevertheless, under large parallelism conditions, such a data structure becomes the bottleneck. Authors in [33] address the efficiency of three CEL implementations, namely the heap, the splay tree and the calendar, and they conclude that the performance of the CEL concept remains mitigated when thousands of concurrent processes access that structure. Therefore, the CEL implementation needs to be parallelized to cope with the parallel architecture of heterogeneous computing resources. The concurrent priority queue [134, 38] is a relevant solution to access and manage the CEL in parallel. An event list or message queue is usually distributed to each logical process in a PDES with its own local clock. The concurrent insertion and deletion of the priority queue, by involving mutual exclusion or atomic functions, leads to the improvement of the overall performance using a global event

list [122]. In the same sense, Chen *et al.* propose a distributed queue which considers multi-core CPUs [27]. However, the above mentioned mechanisms, mutual exclusion and concurrent priority queue, are target-dependent, they could not be directly applied to GPU targets. A different approach is proposed in *et al.* [100], which relies on a hybrid time-event driven simulation based on a GPU-oriented CEL concept that uses a linked list implementation. Despite the fact that this approach has been developed with the aim of improving the GPU-based simulation, the overhead of managing a large number of parallel events remains an open issue due to a limited number of concurrent access to the same physical memory.

5.3.2 Event Scheduling

In the DES and PDES, a large portion of the overall simulation time is used for the events scheduling [102]. Moreover, the efficiency of the scheduler also depends on the synchronization method [84]. The conservative approaches prohibit out-of-order event execution, which in most cases is based on the lookahead concept to preserve the causality rule [46]. Parallel event scheduling is separately studied for a multi-core CPU target [139, 36] and GPUs [100]. Both approaches use a central event queue and several independent threads to fetch the next event from the queue. The multi-threading approach is also applied to reduce the scheduling cost and increase the simulation efficiency but only for a limited number of cores (4 and 8 respectively). Authors in [99] propose a dedicated GPU scheduler based on the SIMD programming model where the event queue is split into several sub-queues to avoid central bottlenecks. All the above mentioned approaches regard a GPU core as a CPU core, which in turn reduces the achievable gain.
The optimistic approach allows an out-of-order execution while ensuring the simulation correctness. Several optimizations were introduced recently to increase the efficiency of optimistic parallel simulations over multi-core CPUs while keeping a reasonable backward compatibility with a standard software architecture [138, 88, 27]. Although such an approach increases the general efficiency by relaxing a substantial limitation of the conservative approach, it introduces a significant memory overhead related to the state-vector saving mechanism, which becomes critical when targeting a very large logical processes.

5.4 Considerations Heterogeneous Computing Considerations

Following the rapid evolution of computing hardware and their capabilities, the sequential simulation approach may not be able to maximize the hardware useage rate to maximize the simulation efficiency. Thus, there is a consensus on the issue: *To increase the simulation efficiency and scalability, weakness of centralized solution needs to be avoided.* On the other side, the computer architecture -that has long remained invariant (CPU-RAM HDD)- has changed radically in the last decade: the

computer becomes a group of heterogeneous resources that collaborate to perform a task. Ignoring that improvement results in a low hardware usage rate which increases the inherent cost of any simulation due to additional resources and power that must be deployed to bridge the waste of resources. To deal with that issue the ingenuity of researchers has proposed a variety of optimizations that we can classify into four classes[110]: *architectural optimization, local optimization, bottlenecks acceleration and hybrid optimization.*

1. The *architectural optimization* attempts to parallelize efficiently 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 [83]. The scalability remains an issue in the flat design when the number of LPs increase [47]. In the literature several optimizations, such as the lookahead [46] and the opportunistic and combined synchronization [110], 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 [95]. 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 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 [144]. 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 [4], where the synchronization and communication overhead are significantly reduced in terms 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 then traditional LPs [106, 26]; thus, the latency issue needs to be addressed.

2. The *local optimization* It aims to improve the efficiency of each LP in its environment. We distinguish two main trends: local parallelism and engineering optimization. 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 available cores to execute in parallel future events[85]. However, this approach relies on a central event list and one scheduler, which remains the bottleneck when targeting larger CPUs (e.g. INTEL MIC with 80 cores) [123]. A dedicated GPU scheduling approach was proposed in [15], 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 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 capability allows a significant performance gain [8]. Nonetheless, that approach is closely related to the implementation of each solution in one side and to the considered hardware on the other side.

3. Bottlenecks acceleration aims to improve the simulation efficiency by reducing the impact of a specific part of the simulation which limit the system performance 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 phenomenon such as the 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. For example, Steenkister et all [19] used the FPGA as a signal accelerator for wireless network simulation. The OpenAirInterface [18] 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 the same context, the GPU emerges as a carrier solution that combines programmability and large computing power. Nevertheless it requires a specific software architecture since its programming model is not fully compliant with the x86 architecture. Despite this limitation, Perumalla et all [108] demonstrate the feasibility of using the GPU as a simulation context and several works prove its efficiency as signal processing accelerator [5, 10]. Other efforts have been given to provide an efficient processing solution based SoC and NOC or even a larger computing solution proposed recently by INTEL [32]. In particular, the XEON Phi co-processor [64] provides up to 64 CPU computing cores per device. That solution seems promising, and several recent works assert that its development-cost/gain tradeoff is interesting [124, 58].

4. In contrast with these fundamental approaches, the hybrid optimization proposes to combine their advantages to reach maximal scalability and efficiency levels. We denoted it as the hybrid category as a reference to its combinatory nature. It introduces a revised software architecture while using massively local optimization and optimized libraries. Moreover, the usage of heterogeneous computing resources is considered to maximize the performance. In this per-

spective, $ns-3$ is a well-known network simulator that combines new software architecture with massively optimized code [76]. Further, new frameworks that rely on virtualized resources combine both architectural and local optimization to perform optimal usage of virtual and real resources [149]. Nevertheless, GPU and hardware acceleration solutions in general are weakly considered in such approaches.

Part II

Contributions

Chapter 6

Cunetsim: An Experimentation Framework to Discover Scalability Horizons

6.1 Introduction

Packet-level network simulators are usually based on a discrete event paradigm where the simulated system is model using a sequence of events. Each event has a discrete timestamps that defines a strict order for the execution. In discrete event simulation, each event is represented by one descriptor that includes the timestamps and execution parameters (i.e. a callback). A simulation framework includes an event scheduler which manages event descriptor and defines the events execution order. In general, such events represent mobility, connectivity, medium modeling (wired or wireless) and in/out packets processing. The time complexity and the used memory of a given simulation are proportional to the number of events. Nevertheless, the number of generated events increases exponentially as a function of both the total number of nodes and traffic load. As a consequence, event management becomes the main bottleneck when targeting large scale simulation. There is also a trade-off between the accuracy of the models, in particular air-medium models (wave propagation), and time complexity that has to be taken into account when targeting large scale simulation.

Accordingly, the parallel and distributed execution appears a trivial candidate since it provides more computing power and memory. However, such an approach introduces new challenges in the management level. In literature reviews, we identify that the communication between different simulation LPs is particularly expensive [48]. In addition, we note that the majority of existing frameworks are based on a distributed approach which ignore new hardware features, despite the fact that the potential speedup of the parallel simulation is clearly identified and proved in several pioneer publications. Investigation research works that draw the limitation of parallel simulation within one context, identify the event scheduling as a major bottleneck which can not be avoided. In fact, even if it is possible to execute several events concurrently, the scheduling process remains centralized which generates the bottleneck of parallel simulation. Several optimization works propose innovative approaches to parallelize the execution. However, the inherent barriers of the DES concept requires passing through a central path for all events.

In contrast with existing approaches, we propose (1) to generate parallel events

rather than detecting their possible concurrent execution and (2) to schedule parallel cloned events only once. These two conceptual modifications are the major contribution of this chapter. The second innovation that we introduce is how we plan to implement these concepts. In fact, we propose to use the GPU as the main simulation support while the CPU acts as the master of the simulation. We target to use Nvidia CUDA GPUs based on the rich development ecosystem around. We define the simulation model according to four points:

1. The unitary component of the simulation is the node, any simulated entity is obligatory relative to a given node.

2. The event generator generates the same event for all nodes. This means that at a given time, all nodes will execute the same event.

3. Each node will be executed in one independent GPU Core. Thus, we guarantee a concurrent execution.

4. The event descriptor is not relative to one node but to all nodes. As a consequence, the event scheduler handle one entry.

We expect that this model will overcome the limitations of traditional DES relative to the scheduling. As a proof of concept, we propose a new CPU-GPU co-simulation framework denoted as Cunetsim, CUDA Network Simulator. Cunetsim is an experimental simulation platform allowing rapid validation and evaluation. In contrast with previous works, Cunetsim is designed to provide an independent parallel execution environment for each simulated node. Nodes communicate through the message passing based on the buffer exchange. Thus, the framework avoids the usage of global knowledge and increases the parallelism level. It is based on the master/worker model for a CPU-GPU co-simulation and provides hybrid synchronization model which maximizes the efficiency and guarantees the correctness of the simulation.
This chapter is structured as following: the second section presents an overview of the four fundamental concepts that direct the design of cunetsim. Section 3 presents the software architecture and features. Section 4 presents a comparative evaluation that highlights the performance of cunetsim compared to existing simulation frameworks. Section 5 gives an overview of several technical challenges that characterize a GPU simulation compared to CPU one. Section 6 summarizes two configuration issues that impact dramatically the performance of the simulation and must be considered when targeting a GPU-oriented simulation. Finally, section 7 concludes this chapter.

6.2 Fundamental Concepts

The goal of cunetsim is to achieve a large scale simulation as fast as possible. The idea of executing the simulation on the GPU seems promising but the usage of such computing power remains delicate due to the required software design which differs

from the traditional x86 code. To harmonize GPU requirements with network simulation specificity, cunetsim framework is built around three fundamental concepts: *the workers pool*, *the separation between the event and the event descriptor* and *the massive parallel events generation.*

6.2.1 The Worker Pool

In master/worker model a worker is assimilated to be an LP that manages a part of the simulation. In general, the worker is still in the same machine during the execution. In the GPU, it is impossible to allocate one process to each computing core and then feed it by events. Thus, we consider that a worker represents an elementary simulated entity (defined by data+ process+ FSM). The workers pool models the total number of simulated entities and share available resources.

6.2.2 Separation Between an Event and Its Description

In DES parallel event scheduling is a permanent challenge which is largely addressed in the literature reviews [45, 82, 127]. Under large scale conditions the event scheduling becomes one of the main bottlenecks. Moreover, detecting events independence in order to schedule them in parallel requires sophisticated algorithms. Nevertheless, whatever the algorithm used to achieve the event scheduling, its cost remains proportional to the number of events. As a consequence, we got the idea of extending the $1:1$ relationship between an event and its descriptor to $n:1$, such that a given event scheduler can handle multiple events using a unique descriptor. This approach disassociates the event number from the scheduling complexity. In fact, if we succeed to overload the event descriptor in order to represent several events rather than just one, then we can bypass the scheduling bottleneck.

6.2.3 Massive Parallel Event 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 cloned events rather than detecting events that can potentially be executed in parallel.
Our understanding of this concept is the following: identical simulated entities may generate the same *event* in order to be executed in the same timestamp. Each event is relative to one entity and will be executed on the corresponding data, attributes and memory. However, with the adequate data representation it is possible to represent these events with one entry on the scheduling system. Thus, the event scheduler handles one entry while execution resources execute several events (as defined by the generator).

6.3 Cunetsim Software Architectures

The software architecture of cunetsim is based on a master/worker model where the master manages the simulation and the workers pool executes the computing tasks [96]. Cunetsim is a proof of concept of large scale network simulation. Thus, the finite state machine defining the worker is based on five states: i) the application, (ii) the protocol stack, (iii) the mobility, (iv) the connectivity and (v) the packet services. The master is composed of five components: (i) the event scheduler, (ii) the data abstraction layer, (iii) the scenario manager, (iv) the monitoring component and (v) the helper . In this chapter, we focus on two components: the event scheduler and (ii) the data abstraction layer. We note that from a software point of view, we denoted the scheduling of a given state at a given timestamp as an event pattern (EP) [18]. This usage is relative to the fact that the scheduler is handling a kind of pattern that will generate several real events at the execution level. Figure 6.2 presents an overview of the event flow between the master and the workers pool. In particular, the figure highlights how the hardware scheduler transforms an event pattern to a group of events, each for one worker.

6.3.1 The Worker Design

The Worker implements the simulated node, modeled as a stack of independent EPs. Nodes communicate through message passing. Only buffers are exchanged between nodes to avoid global knowledge. In Cunetsim, each node contains five ordered EPs.

6.3.1.1 Application (APP)

The application EP models the application and provides a packet-level traffic generator to simulate application data, based on packet size and inter-departure time. Each instance is completely independent, allowing the framework to support an important load. The traffic generator tags the packet as a function of communication type: unicast, multicast and broadcast.

6.3.1.2 Protocol stack (PROTO)

The protocol stack EP represents the internal processing of the communication flow. A realistic implementation of a given protocol stack requires a significant development effort. Nevertheless, such an implementation does not impact the proof of the proposed concepts. Thus, we simplify the protocol stack implementation and provide minimal service to ensure the processing of received packets from outside of the APP. The PROTO event pattern ensures also the packet forwarding operation. Cunetsim implements various broadcasting techniques, such as probabilistic, counter-based and location-based. Such implementations support GPU parallelism.

6.3.1.3 Mobility (MOB)

The MOB EP calculates a specific movement in the defined space following a mobility model, for each node. There is a generic mobility container, implemented as a unique CUDA kernel. Figure.6.1 explains the identification of the mobility routines, based on the id of each node. The preliminary implementation provides two mobility models: *RandomWayPoint* and *RandomDirection* [57] and three boundary policy models: *Annulment of excess*, *Sliding on the boundaries* and *Bouncing on the boundaries*.

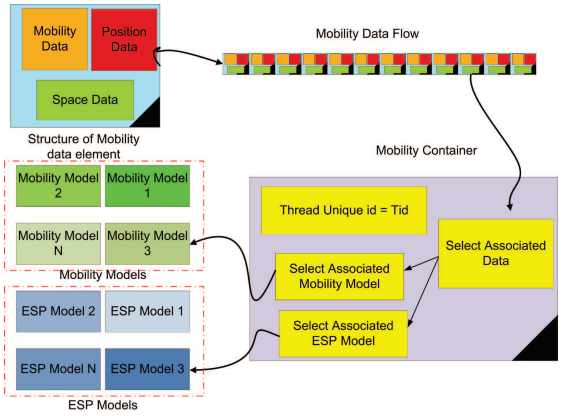


Figure 6.1: Mobility Events pattern

According to the Tid, each worker recognizes its data which in return allows it to select the corresponding model.

6.3.1.4 Connectivity (CON)

The connectivity EP identifies all neighbors of the concerned node. This problem is NP-Complete[23]. The complexity of the brute force approach is of the order of $O(N^2)$ in 2D space. In Cunetsim design, we divided the space into geometric cells where the radius of the cell must be at least the double of the maximum transmission range ($2*Rmax$). In this case, each node will find its neighbors in its own cell as well as in the neighboring cells. This approach reduces significantly the complexity. We define a connectivity container, which we will call a specific connectivity model (the Unit Disk Graph *(UDG)* or the Quasi Unit Disk Graph *(QUDG)*). This kernel will be instantiated into N GPU threads, where N is the nodes number in the scenario.

6.3.1.5 Packets services (PKT)

The packets-services EP manages the packets exchange between nodes. The notion of packet can represent any protocol data unit (PDU), which is layer-dependent. To simulate multiple interfaces, a node may have more than one buffer, each of which

is associated with a given interface. Packet services support both *send* and *receive*. *Packet-send* service allows a node to write a packet to the selected in-buffer of the neighbor(s). *The write* operation is atomic. *Packet-receive* service allows a node to read at most one packet from its in-buffer at each simulation time (i.e. round). However, the in-buffer is capable of receiving up to M packets from other nodes at each round. The receiving service determines which message has to be read by the node based on the lowest timestamps and/or channel quality estimation.

6.3.2 The Master Design

The master ensures the simulation correctness, simplifies the framework usability by providing high-level simulation APIs, and guarantees the simulation reproducibility. To reach these goals, it relies particularly on two main components: the events scheduler and the data abstraction layer.

6.3.2.1 Cunetsim Events Scheduler

Cunetsim events scheduler (CES) implements a conservative approach for all dependent EPs according to a strict order between sequential EPs for each node. This model was developed in[15] where the notion of *EP Pool* is introduced: a EP pool, Π_i incorporates all event E_j relative to all nodes that must be executed at $T = i$. For a given Π_i, all E_{ij} must finally assert that Π is achieved. This presents a simple yet efficient implementation of the coherence and consistency paradigm. In addition, CES incorporates an opportunistic mechanism for independent EPs having the same timestamps. In fact, independent EPs concern typically heterogeneous components and occur when nodes are composed of different EP sequences. In such case, conservative approach does not define a deterministic order according to the timestamp.
To summarize, the Cunetsim hybrid event scheduler works as follows: The conservative approach is used for sequential EPs as defined by the simulation model. The optimistic approach is applied when there are several events pools having the same timestamps. It has to be mentioned that the CES benefits from the GPU hardware scheduling capabilities. Indeed, the optimistic approach is achieved using the GigaThreads scheduler (i.e. GPU hardware acceleration), the relaxed approach using the wrap schedulers of each SM. The conservative approach is implemented in software.
Figure 6.2 models a simplified adaptation of the master/worker model for a mono-GPU software architecture. The master handles grouped events and delegates the creation of cloned events to the GPU Giga-thread-scheduler[1]. Furthermore, that

[1]One of the most important technologies of the Fermi architecture is its two-level, distributed thread scheduler. At the chip level, a global work distribution engine schedules thread blocks to various SMs, while at the SM level, each warp scheduler distributes warps of 32 threads to its execution units. The first generation GigaThread engine introduced in G80 managed up to 12,288 threads in realtime. The Fermi architecture improves on this foundation by providing not only greater thread throughput, but dramatically faster context switching, concurrent kernel execution,

concept relies on the hardware synchronization features.

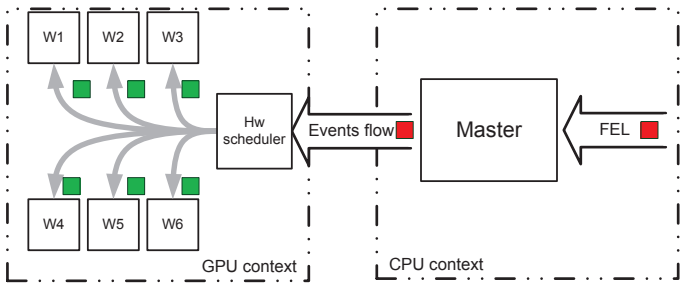


Figure 6.2: Simplified master/workers model that targets GPU execution.

The hardware scheduler accelerates significantly the scheduling task.

6.3.2.2 Data Abstraction Layer

Cunetsim data are modeled based on the kernel/flow model. We define several flows where each one presents a specific part of the simulation data. Data is grouped by functionality. Each EP uses one (or more) flow(s) and each node has a specific buffer with R/W rights. One node can access foreign data with read right. Flow model is natively used by graphics application to manage the communication between the GPU and the CPU. We apply a flow loading-offloading mechanism between the GPU memory (limited and non-extensible) and the principal memory (larger and extensible). The master manages the transfer of data flows between the principal memory and the GPU one, such that no EPs will be in a famine situation.
The data abstraction component provides two services: memory allocation abstraction (MAA) and critical section management (CSM). MAA insures the double allocation of each data flow in both the RAM and the DRAM. The synchronization of both copies of the flow is a manual operation which must be specified by the user. Critical section is a recurrent challenge in case of shared memory between several processes. Software mutual exclusion such as semaphores, mutex and locks are commonly used in CPU context. However, GPU context does not provide such explicit solutions. The problem arises mainly when two nodes try at the same time to write messages in a third node's buffer. CSM provides an abstraction of this problem based on CUDA atomic operations: thanks to atomicInc, a node makes an atomic reservation operation before proceeding to the writing of the message.

6.3.3 Legacy Architecture for Multi-Core CPU

The master/worker architecture that we present in section 6.3 is natively adequate for the GPU architecture. However, the GPGPU is a recent discipline and rare

and improved thread block scheduling. [104]

are the data-centers which use the GPUs as computing co-processors. Furthermore, current and future CPUs are also multi-core and provide interesting features, such as vector parallelism. Accordingly, it seems primary to provide a CPU version which will validate the efficiency of the proposed concepts under fair conditions. Nevertheless, it is mandatory that both version (CPU and GPU) share the same models, design and implementation. Thus, we use the PGI unified binary technology [3] to create a CPU binary from the GPU source code.
While the legacy architecture is directly derived from that targeting the GPU, it presents two particularities: first, it uses a software API to create different events from cloned ones; secondly, it does not require a data transfer and can use a large data set. Figure 6.3 presents a typical legacy architecture that uses the OpenMP API as an event dispatcher. In the remainder, the legacy version will be refered to as Cunetsim-LN, where N represents the number of workers.

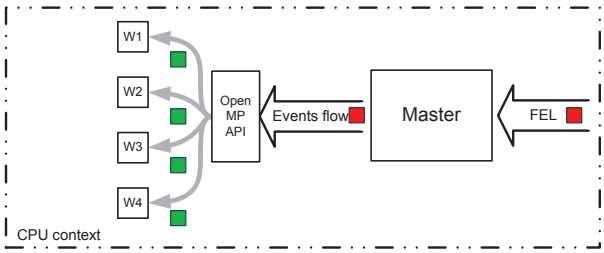


Figure 6.3: Simplified master/workers model that targets CPU execution.

From a user/developer point of view, the creation of different workers is transparent, the OpenMP API ensures this task.

6.4 Comparative Performances Results

To evaluate the performance of the proposed design under large scale conditions regardless of different models impact, we extend the benchmarking methodology for network simulators presented in[143] to support wireless and mobility. In this methodology, authors implement identical node models for all considered simulators. They show that NS-3[2], Jist and Omnet++ have the best performance. However, the benchmarking model does not address mobility and wireless issues. Furthermore, the comparison does not include sinlago [1], known as a stable simulator on large scale conditions. In the following study, we have chosen Sinalgo as a representative framework of CPU-based solution while NS-3 is involved as the most optimized open source simulator, providing also a stable distributed version over MPI. The CPU version of cunetsim, which involves 4 CPU cores is a representative case of the CPU implementation of the fundamental concepts discussed in section 6.2. The mobility and connectivity algorithms are the same for all simulators. Only a simple flooding

protocol is implemented using equivalent algorithms.
We propose two benchmarking scenarios: The first compares the performance of each simulator's kernel regardless of the efficiency of implemented models, while the second addresses their robustness in mobile conditions. The first scenario models a simple static network, where the nodes are arranged in a grid topology as illustrated in Figure 6.4. The scenario model includes one traffic source which generates 600

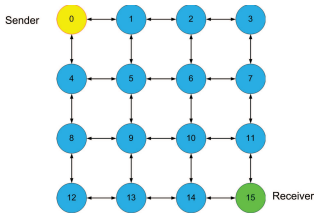


Figure 6.4: Simple Grid Topology

The traffic source is the node with the lowest id, the receiver is the node with the highest id, the intermediate nodes forward received packets according to the channel description.

uniform packets with 1 second of inter-departure time. Packet size is fixed to 128 bytes. All nodes relay unseen packets after a delay of 1 second, thus flooding the totality of the network. The delay of 1 second models the propagation. Nodes do not provide any packet management services. Transmission and reliability are modeled on the channel using a dropping probability which is identical on all links. The sender is the node with the lowest identity and the receiver is the one with the highest identity.
In the second scenario, nodes are mobile. The mobility model is the random way point with speed uniformly distributed between $1 - 5m/s$. The maximum transmission range, Rmax, is 100 and the connectivity model is UDG. The simulation space is a cubic free space whose dimensions are $1600 * 1600 * 1600$ m. Each node moves before each round and recalculates its connectivity set.
Both of these scenarios are outlying real networks and include major node simplifications. Nevertheless, they have two advantages: they guarantee a relevant and neutral comparative since they minimize the models impact and they provide a representative estimation of the computing power needed for such simulations. All simulation runs are conducted using a simple PC including an INTEL i7 940 CPU (4 cores with hyper-threading), 6GB of DDR3 and one GPU: the GeForce 460 1GB (336 cores for GPGPU computing). The OS is Ubuntu Linux 11.10, the Java version is 1.6 and the Nvidia driver version is 285.05.33. Our measurements were taken using NS-3.13, Sinalgo 10.75.03 and Cunetsim prototype.
To validate the model equality, we use the first scenario with all simulators where we vary the drop probabilities in the interval $[0, 1]$. Figure.6.5 depicts the end-to-end packet loss repossessed and normalized from different simulators, given the dropping

probability and the network size. All studied simulators produce similar results. We conclude that our implementations are equivalent -in terms of output- to those of [143]. We evaluate simulators' efficiency regarding the simulation runtime. Our results give the average of five executions. The minimal simulation time is set to 700 seconds.

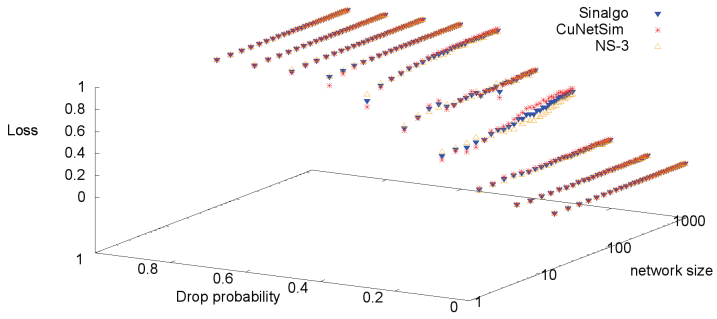


Figure 6.5: End-to-End Packet Loss:

Under the same conditions, all simulators present equivalent E-TO-E loss, considered as the output.

6.4.1 Simulation Runtime

To evaluate the simulation runtime of $NS-3$,sinalgo, and both cunetsim versions, we have fixed the drop probability to 0.1 and we have increased the network size from 4 to $102K$ nodes. We analyze the measurement relative to the first scenario in section 6.4.1.1 and that relative to the second one in section 6.4.1.2.

6.4.1.1 First Scenario (static topology)

Figure 6.6 shows the average simulation runtime for each simulator. For small to medium networks, Cunetsim-L4 is the fastest simulator up to 2000 nodes and remains faster than Sinalgo and $ns-3$ in all cases. Beyond, Cunetsim (GPU) becomes the most efficient simulator and the deviation is growing with the network size. As a function of the simulators runtime, we distinguish four network size intervals: small networks [2-50], medium networks [50-200], large networks [200-2000] and the very large networks [2000-102000].
For the small scale, Cunetsim-L4 and $ns-3$ are the most efficient simulators. In fact, both simulators use the very high-speed L3 cache compared with slower RAM. Cunetsim-GPU is outperformed due to two reasons: first the latency of data transfer between the RAM and GDRAM is significant, second the GPU is underused since

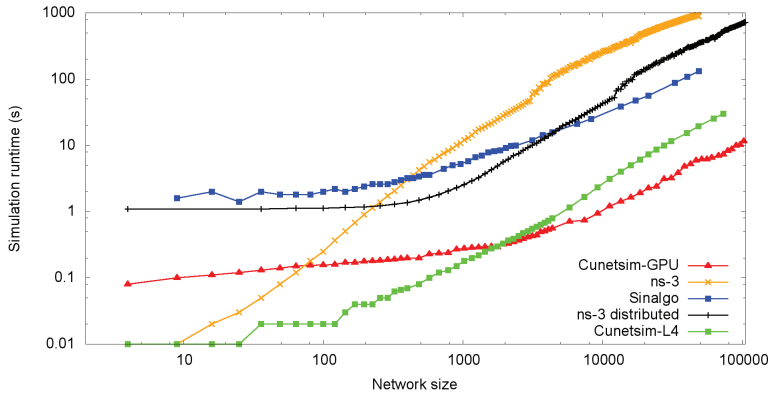


Figure 6.6: Simulation runtime of the static network:

CPU simulators are efficient under small scale conditions. however, their limited computing power appears as major handicap under large scale conditions.

only few cores are active (the number of workers is less than the number of available cores).

For medium scale, both versions of cunetsim are faster than $ns-3$S. In fact, when the network size increases, cunetsim uses additional GPU cores. However, the data transfer between the RAM and the GDRAM remains significant which allows Cunetsim-L4 to be the fastest solution. In both intervals, distributed NS-3 suffers from the initial setup load of the MPI, relegating it behind the classic version.

As the number of nodes increases, sinalgo outperforms ns-3 thanks to its optimized nodes management. However, the distributed version of $ns-3$ remains stable overwhelming easily sinalgo. Furthermore, cunetsim-L4, reaches the CPU limit while Cunetsim-GPU remains stable in this interval. Finally, from the threshold of 2000 nodes, both of Cunetsim-GPU and Cunetsim-L4 need 0.35 second, 80 times faster than $ns-3$ and 26 times faster than Sinalgo.

For very large scale, the power of GPU is revealed, the number of cores involved in the simulation provides a significant computing power in contrast with the limited number of CPU cores. Accordingly, cunetsim-L4 cannot rivals with the GPU version even if it remains the most efficient CPU-based simulator. Thus, for 48K nodes cunetsim needs 5.93 seconds, 3.5 times faster than cunetsim-L4, 22 times faster than sinalgo and 150 times faster than $ns-3$. It is interesting to compare in such scale Cunetsim-4L and the distributed $ns-3$ since both use the same computing power in theory. In fact, Cunetsim-4L overcomes NS-3 due to two major reasons: first, it uses a shared memory synchronization (over OpenMP) while NS-3 uses an important protocol stack (over MPI). Secondly, the events scheduling is completely different: Cunetsim has a prior knowledge regarding events relationship while NS-3 has only their timestamps as a scheduling information.

6.4.1.2 Second Scenario (mobile nodes)

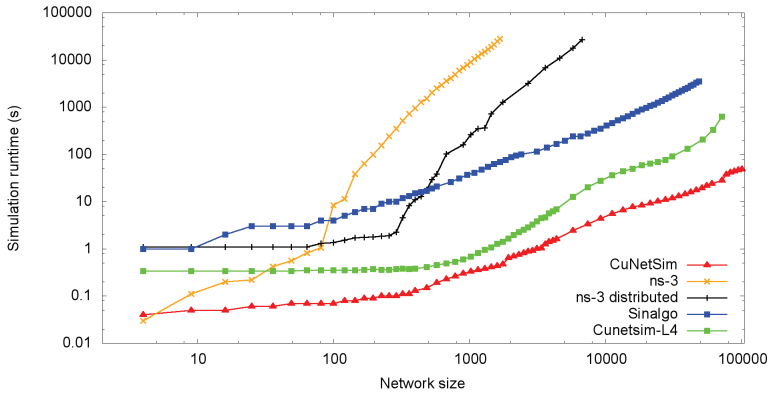


Figure 6.7: Simulation runtime of the mobile network:

The complexity of the wireless mobile scenario highlights the limitation of classic approaches under large scale conditions.

Figure. 6.7 shows the average measured simulation runtime for each simulator. The mobility causes link connection and disconnection according to the relative position of each node. Such an operation is extremely expensive in terms of computing resources. Thus, combining distributed software architecture with a massively parallel computing support such as the GPU reveals its efficiency under these conditions. Accordingly, except for very small networks (4 nodes) the GPU version of cunetsim is definitively the most efficient solution in term of scalability (able to reach 100k) and running time. $Ns-3$ is the fastest CPU-based simulator up to 36 nodes and cunetsim-L4 becomes the fastest CPU-based simulator beyond. NS-3 runtime increases exponentially as a function of network size while Sinalgo and Cunetsim-L4 present stable behavior. The distributed version of NS-3 increases its leeway but does not influence the global behavior. Thus, distributed NS-3 remains faster than sinalgo up to 800 nodes but its computing time becomes unstable further. Cunetsim-L4 runtime remains relatively invariant for small to medium networks, and becomes a function of nodes' number nearby of 1000. Sinalgo presents a quasi-linear runtime as a function of the network size but cannot achieve a very large scale simulation in realistic time (simulating 48K nodes requires 3552 seconds). Cunetsim runtime is linear per segment between 4 and 8000 nodes. From this threshold, it becomes relative to the network size but remains reasonable, even for $100K$ nodes. In all cases, Cunetsim is extremely faster than all CPU-based simulators. Thus, for $48k$ nodes cunetsim is up to $9.2x$ faster than Cunetsim-L4 and 260 times faster than sinalgo. NS-3 is unable to compete in such scale. In addition, the CPU-legacy version presents very interesting results since it is 28 times faster than Sinalgo for the same scale. According to distributed $ns-3$ results, distributing the simulation over several LPs to maximize the usage of multi-core CPUs is not sufficient.

6.5 Technical Challenges of GPU-based Simulation

In this chapter, we demonstrate that the usage of the GPU is capable of providing opportunities in terms of simulation efficiency. However, the usage of the GPU requires an important software modification and tuning in order to achieve the target efficiency. Several works propose optimization principals to cope with GPU specificity [91]. During the development of Cunetsim, we face several technical challenges that present significant impact in the correctness and the efficiency of the simulation. In this section, we summarize three major challenges that must be considered when designing any massively parallel simulation framework: the synchronization, the memory management and the precision issue.

6.5.1 Synchronization Challenge

In sequential simulation, each logical process has its referential clock that governs the time advancement. All simulated entities refer to that clock in order to compute or execute any operation. When targeting parallel simulation on multi-core CPU context, there are several primitives which allow the synchronization of executed processes. In event driven simulation, each event is supposed to be atomic. The simulation is suspended until achieving the current event. Parallel DES provides equivalent features for each executed event with data protection mechanisms.
The procedure is different when we target the GPU as an execution context. Let us have an overview of the parallel execution on the GPU. To execute cloned events, the user provides three elements: the event implementation, the event multiplicity representing the number of instances that user wants to launch and the relative data, typically represented by a vector. The event multiplicity is defined by two 3-dimensional parameters: the block and the Grid. The block value represents the number of threads that will share the same SM during their execution. The grid value represents the number of blocks. Assume that we use $1D$ parameter, thus, we have a vector of thread per block and a group of independent blocks.
The CUDA programming model assumes that threads of the same block will be executed *as good as possible in parallel* and assumes that there is *no guarantee* about block order[2]. From the masters point of view, before launching the grouped entry corresponding to cloned events, the timestamps were $T1$ and after its execution it will be $T1 + e$. From workers' point of view, each one starts executing the event at $T1$ and finishes at $T1 + e$. However, at a given real time during the execution, it is possible that some workers have their clock at $T1$ while others are at $T1 + e$ (note that workers are concurrent on the available resources and that the number of workers is much more important than that of GPU cores). Accordingly, during the execution of an event, workers are inevitably non-synchronized.
Depending on the type of the event, the non-synchronization may impact or not the simulation correctness. For example if the event is totally internal such as trans-

[2]If you have 6 blocks that must be executed on 3 SMs, there is no guarantee that blocks 1,2 and 3 will be executed before 4,5 and 6.

ferring a packet from layer N to layer $N + 1$ or executing the battery updating routine, the user does not care about which worker finishes first. However, if the event requires the usage of shared data, then the software designer must specify access and protections rules and may use synchronization techniques. In the following sections, we propose two basic techniques that can resolve the event synchronization challenge.

6.5.1.1 Physical threads synchronization

There are two levels of hardware threads synchronization: the block and the grid. At the block level, the CUDA API provides an interesting primitive (__synchthreads()) that forces the synchronization of all threads of the same blocks. The __synchthreads works as a mandatory checkpoint for all threads. On the grid level, it is conceptually forbidden to define a specific order between blocks. Thus, the safest method consists of dividing the corresponding events into two logical subevents, each of which performs a part of the event. In this case, the host process creates the physical checkpoint.
These methods guarantee the simulation correctness. However, they introduces a significant waste of resource. In cunetsim case, we observe performance degradation that may reach up to 50% when we rely on such mechanisms to ensure the workers synchronization. This is what justifies that there is a need for a simpler synchronization that avoids the usage of standard synchronization methods.

6.5.1.2 Atomic Operation

If the synchronization issue is relative to data usage, then it is possible to use atomics operations to manage critical memory access. The general concept is to provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads. This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates. Atomic functions perform read-modify-write operations on data residing in global and shared memory. Atomic functions guarantee that only one thread may access a memory location while the operation completes. The hardware ensures that all statements are executed atomically without interruption by any other atomic functions. The atomic function returns the initial value, not the final one, stored at the memory location. Its success means that the concerned variable includes the new value, not the returned one. The order in which concurrent atomic updates are performed is not defined and may appear arbitrary. However, none of the atomic updates will be lost. Concerning the CUDA API, there are different kinds of atomic operations:

- Add (add), Sub (subtract), Inc (increment), Dec (decrement)
- And (bit-wise and), Or (bit-wise or) , Xor (bit-wise exclusive or)
- Exch (Exchange)

- Min (Minimum), Max (Maximum)
- Compare-and-Swap

Nevertheless, these atomic operations do not provide a sophisticated mechanism that lock large memory zone which is mandatory in any simulation. Thus, we have created several structures that imitate the behavior of a critical section with minimal impact on the performance. For example, the implementation of LIFO buffer relies on one integer index. At the initialization phase, the index is set to zero which means that the buffer is empty. Each time that a worker wants to write data into that buffer, it increments the index atomically. The read value is the index of the empty frame where a worker can write. Any other worker that also wants to write data in the same buffer will read the next value. On the other side, the consumer uses the atomic decrementation if there is no confusion. Otherwise, the consumer must lock the buffer in order to read without interference. The easiest way to lock is to put the index to its maximal value which blocks writers. Flexible LIFO buffer allowing simultaneous reading and writing in different frames is managed by two indexes: reading and writing.

6.5.2 Memory Management Challenge

In contrast with traditional programming model where the data management relies on a unified addressing space physically located on the RAM, the GPU programming model requires specific attention on data and memory management. First, there are at least two memory spaces: the CPU and the GPU(s). Generally, users must allocate the required memory space in both. After an initialization phase on the CPU space, the data will be transferred to the GPU context. If this operation is limited to initialization and termination phases, the overhead remains reduced. However, it is common that the user needs a continuous updating for supervision and monitoring tasks. Accordingly this transfer becomes a significant overhead that must be optimized. The second issue is the different memory levels on the GPU itself (global, shared, register, texture, constant). While traditional CPU compilers are mature enough to maximize the usage of the L1, L2 and L3 caches without a specific attention from the developer, GPU compilers do not curently offer such a behavior. In fact, a GPU compilers handle complicated software architectures where it is difficult to predict threads interaction based on the code. Accordingly, the user must specify which data will be cached on each memory stage.
To deal with the first issue, Cunetsim framework provides a unique API that manages both memory allocation and simplifies the data transfer between them. Nevertheless, the real challenge relies on the significant difference between both sizes. In fact, the RAM is natively extensible and larger than the GRAM. Thus, when the required memory is larger than what the GRAM can provide, we use the notion of stream. A stream is dynamically loaded and offloaded to the GRAM according to the processing advancement.

6.5.3 Precision Issue

The implementation of the floating point on NVIDIA device is not fully IEEE compliant. To analyze the difference between a GPU and CPU implementation, we use the distance computing between two nodes as a benchmark test and calculate this distance using both, over 1 million of samples. The difference between each pair of results is less than 0.01%. Depending on the scenario, this difference might cause some simulation inaccuracy (e.g. channel realization).

6.6 Configuration Issues of GPU-Oriented Simulation

In addition to technical challenges that we presented in section 6.6, parallel and distributed simulation over GPU imposes several configuration issues that influence the general performance. In this section, we will present two representative issues: the space representation and the block size as a study case of tuning parameters.

6.6.1 Space Representation and partitioning

In distributed simulation, the partitioning of the simulated environment is always challenging and influences seriously the performance [125, 31]. In parallel simulation based on shared memory, the issue seems less critical in CPU context. In fact, environment data such that space is shared between different execution processes. Hence, when the number of concurrent processes is reduced (between 4 and 8), environment information can be shared and protected. However, in massively parallel context where thousands of independent processes are executed simultaneously, locking environment data for any modification becomes extremely inefficient. The solution that we adopt is to divide environmental data into several independent and contiguous frames so that the concurrency between processes will be reduced to a reasonable number. To apply this approach to Cunetsim, we divide the simulation space into contiguous geometric cells that respect fives rules:

- A worker evolves into one cell at a given time.
- A worker can interact with elements of its cell and that of neighboring cells.
- A worker can move from one cell to another.
- Each cell keeps the information about any present node.
- Locality information is saved on the fastest memory.

6.6.2 Tuning Parameters: Block Size as a Study Case

One embarrassing phenomenon that concerns the GPU programming is the sensibility to both hardware and software tuning parameters. In fact, it is difficult to predict the behavior of a given code on GPU without profiling operation. Moreover, defining analytically the best configuration seems unrealistic. One significant

tuning parameter that impact the simulation performance is the block size: how many threads will we put together in order to be executed on the same SM? In fact, a hardware scheduler is associated with each SM. It decomposes the block into warps so that threads of the same warp have the same code path. A warp is executed in parallel, but warps of the same block are concurrent. The SM scheduler manages the execution of warps and interrupts the execution of one to launch another with respect to internal algorithms that aim at maximizing the hardware usage rate. CUDA specification suggests that the number of threads' block must be four times greater than the number of execution core on the target SM to reach maximal efficiency. However, from a GPU generation to another, the specification of the SM changes. Currently the number of execution cores varies from 32 to 192. Furthermore, the GPU architecture defines a maximal block size (between 512 and 1024). Our experimental measurement shows that there is no optimal value for all configurations.

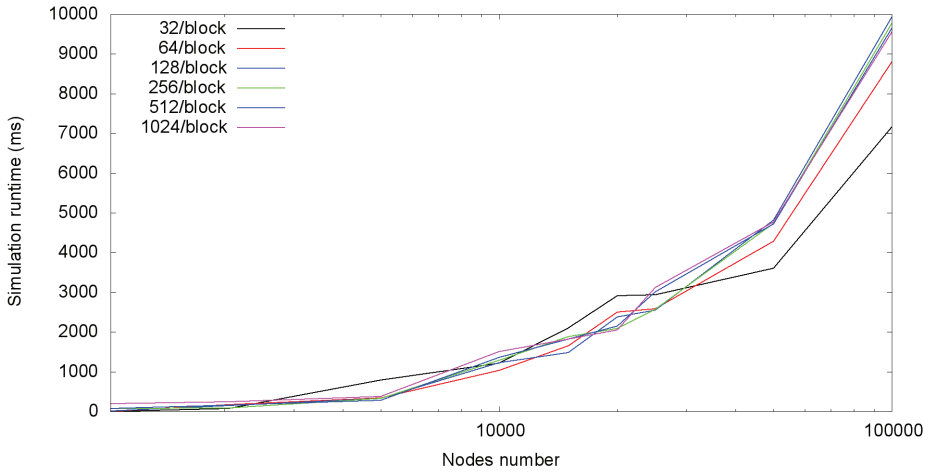


Figure 6.8: Block size impact

According to the number of threads per block and the number of nodes (total number of threads), the performance of the simulation radically changes. Thus, for very large scale, 32threads per block seems the most adequate configuration.

To highlight the impact of the block size, we use a referential scenario that relies on wireless mobile nodes. We increase the number of nodes from 1024 to 100 K nodes in order to increase the simulation load, and we repeat the experimentation with six sizes: 32,64,128,256,512 and 1024. Figure 6.8 shows the simulation runtime of the experimentation series. First, we note that there is no optimal configuration, each size presents punctual superiority. Secondly, we note that for a given simulation load, the difference between the maximal and the minimal value is about 20%. Accordingly, we propose two solutions to maintain a reasonable trade-off between efficiency and decision complexity: the dynamic update and the static referential

table.

Dynamic update: the principal of the dynamic update is to detect the most adequate block size according to the simulation environment retrospectively. Thus, the master process increases the block size continuously from 32 to 1024 in order to detect the optimal configuration. The master iterates this operation periodically to adapt the block size to the simulation behavior. The dynamic update method introduces a small overhead since it relies on real measurement during the simulation but may approach the ideal configuration.

Static Referential Table: generally, the number of different GPU architecture on the same testbed is reduced (1-3). Thus, it is possible to determine apriority the best configuration for each couple (GPU-event) and store this information into a static table. This method does not interfere with the simulation and allows the user to approach a good configuration.

6.7 Conclusion

To address the scalability issue, we propose to consider two limitations: the computing power and the event scheduling complexity. Thus, we propose to use the GPU as a main simulation context to take advantage of its computing power. To cope with the GPU specificities, we propose to generate cloned events that share the same event code and to group them into a unique entry that will be handled by the event scheduler. As a proof of concept, we propose to create a new simplified network simulator that aims at validating the pertinence of these proposals. Furthermore, we analyze the pertinence of dissociating between the event and its descriptor during management phases (generation and scheduling).

Finally, Cunetsim aims at unlocking the parallel capabilities of the state-of-the-art hardware and software architectures to achieve simulation scalability and efficiency with a significantly lower cost. Cunetsim is a fully GPU-based simulator which provides a CPU-GPU co-simulation framework for large-scale scenarios.

In contrast with existing GPU acceleration approaches, the simulation is fully executed on the GPU. Furthermore, Cunetsim proposes an efficient solution to manage critical sections which presents a real challenge of the GPU programming model. Performance results show that the runtime could be improved when GPU parallelism is used to carry out the simulation. In particular, Cunetsim is able to achieve up to 260 faster than existing simulators, when targeting large scale mobile networks. Nevertheless, based on the cunetsim experience, we have two critical observations: first, it seems inadequate to achieve a general purpose simulation exclusively on the GPU context due to the memory limitation on the one hand and to the obligation of event grouping, on the other hand. Secondly, achieving a very large scale simulation requires a distrusted infrastructure -at least to provide the necessary memory space- where the simulation context and the simulated population are distributed on distinct machines. Both issues are discussed in the following chapters.

Chapter 7

Hybrid Events Scheduler

7.1 Introduction

Discrete event simulation (DES) is largely used to model, analyze, and evaluate complex systems, where formal analysis is difficult or non-deterministic. However, the scalability of DES remains challenging due to the increasing system and model complexity on the one hand, and the phenomenal inter-connectivity features of recent systems on the other hand. In addition, one fundamental limitation of current DES is the absence of a dedicated event management policy that considers heterogeneous computing capabilities. In that context, event scheduling is identified as an inherent bottleneck of DES. In particular, scheduling the execution of future events while maintaining a continuous load under large-scale conditions increases the scheduling cost, until it becomes the bottleneck [47]. Most popular scheduling approaches rely on a centralized event scheduling model optimized mainly for the homogeneous computing node architecture. Such a model remains limited and does not exploit the full modern hardware potential. Hence, the parallel and distributed scheduling approaches reemerge as a major factor to increase the scalability over heterogeneous computing architectures [135]. The objective is to exploit a multitude of parallel and interactive processors unified at the level of event scheduler to cooperate with each other. Examples include multi-core CPUs, multi-GPUs, multi-processor system-on-chip and accelerated processing unit. Regarding this requirement, we highlight the need of a verified scheduling framework that combines hardware abstraction with simplified management.

Most of those architectures seem promising, but their ecosystems in some cases are either not fully developed or conflicting [21]. Benefits of GPU-enabled supercomputers have been highlighted in [108], where authors suggest revisiting and expanding the vision of DES. Nevertheless, most of the recent attempts assume the backward compatibility with the sequential scheduling concept [99, 139]. Such a methodology presents a conceptual weakness since it considers a multi-core computing node as a simple extension of a mono-core one. Furthermore, to remain backward compatible, the expected gain will be significantly reduced when compared to a dedicated software design which exploits the parallel computing capabilities of the current hardware as well as the communication latency [4].

In this work, we introduce a new parallel event scheduler for heterogeneous computing architectures, denoted as Hybrid scheduler (H-scheduler). The H-scheduler is designed to dynamically allocate events to available computing resources while keeping a stable event rate. This is achieved as the scheduler is aware of the heap

of processors and their capabilities and has a constant access to their instantaneous loads and execution time through a feedback mechanism. The scheduler operates on all the available computing resources within the same addressing memory space. To increase the scheduler efficiency, each event is associated with a specific descriptor that will be stored into a 3-dimensional data structure.This 3D data structure allows the framework to cope with consecutive timestamped events, isolated/recursive events, and cloned event. Since the objective of this work is to maximize the efficiency of the simulation, first attempt was to use opportunistic scheduling policies. However, we decide to use the conservative scheduling policy to avoid the overhead generated by the recovery mechanism and the state vector when considering optimistic policy in parallel and heterogeneous settings.

The H-scheduler is composed of four main processes: event dispatcher, event injector, GPU-scheduler, and CPU-scheduler, where events are flowing.

1. *The dispatcher* fetches the newly generated events from different queues and adds them to a corresponding position within a 3-dimensional data structure optimized for the parallel execution.

2. *The injector* directs a group of parallel events to the most adequate sub-scheduler (CPU or GPU) based on the received feedback information.

3. *The GPU-scheduler* ensures the execution of grouped entries on the dedicated GPU.

4. *The CPU-scheduler* ensures the execution of any switched entry on the dedicated CPU.

Each sub-scheduler is optimized for a specific hardware in order to maximize the activity rate of the corresponding computing resources. Several optimizations are proposed to accelerate the scheduling decision as the bottleneck may change over time. The H-scheduler mechanism relies on three policies for both dispatcher and injector processes: rapid, advanced and hybrid. The rapid policy aims to minimize the decision cost while the advanced one aims to optimize the execution target according the hardware load. The hybrid policy uses both advanced and rapid policies to maximize the stability of the system. Comparative assessments have demonstrated that the performance gain could be increased by a factor of 2 compared to centralized and conservative schedulers.

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

7.2 The Hybrid Scheduler

To operate on a heterogeneous computing node, the hybrid scheduler relies on three fundamental concepts: (A) the event descriptor, (B) the event structure, and (C) the event flow.

(A) The event descriptor extends the traditional event metadata, namely timestamps, id, in/out data, to support additional information used to reduce the parallel event scheduling cost. In particular, it includes (i) event dependency information, (ii) event execution timestamp, (iii) I/O data access, (iv) event structure information, and (v) execution targets. The event dependency defines if an event has one or multiple dependencies (if it needs current output as input). The event execution timestamp identifies which events can be scheduled in parallel for a given timestamp and is calculated based on the current timestamp and the safety lookahead. The I/O data access defines the permissions given to an event to read and/or write a shared memory area. The event structure information identifies an event as a CIE or a IFE for the given timestamp. Finally, the execution target defines where an event could be executed in the CPU, GPU or both.

(B) The event structure expands how the events are represented so as to increase the parallelism in a heterogeneous computing architecture. In this purpose, events are dispatched over a 3-dimensional arraylist, where each element of the array represents the timestamp and the associated list represents the parallel event sets for a given timestamp interval. The parallel event set is composed of the CIE (SIMD-like) and IFE the (MIMD-like). It has to be mentioned that the CIE are 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.

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

7.2.1 Model and Components

To perform an efficient parallel event scheduling in large scale conditions over a heterogeneous computing machine, the scheduler is designed as a system including several processes and buffers. It includes the future event list (FEL), the dispatcher, arraylists (AL), the injector, the GPU-scheduler, the CPU-scheduler, and the feedback mechanism as shown in the Figure 7.2. The FEL is a standard FIFO providing reliable status flag used to collect and manage the produced events. There is one dedicated FEL per computing resources to gather the generated events and one FEL for all the incoming events. The dispatcher is the front-end of the scheduler, it first

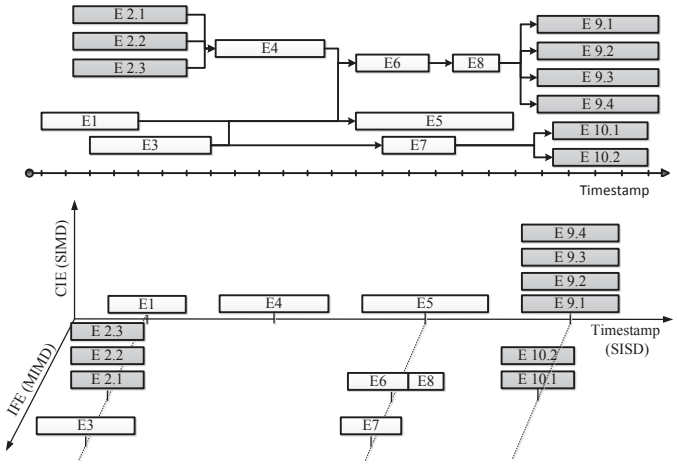


Figure 7.1: Events reordering and storage: top figure schematizes an events dependency diagram while bottom figure schematizes how they will be reordered and stored. Events dependency is transformed to interval.

reads the events from the FEL following a given scheduling policy (e.g. weighted round robin). Then, it adds events into the global 3-dimensional AL data structure based on the event descriptor to ensure the simulation correctness. The injector proceeds on the per-interval basis and dynamically determines the target computing resource, i.e. GPU or CPU, as well as the subset of events to be allocated based on the event descriptor, received feedback information and the target capacities.

As a result, the subset of allocated events will be pushed to the local 2D-AL of a target. Please note that the injector starts the next interval only when all the events associated to the current interval are executed. Both the GPU and CPU scheduler receive events on a dedicated local AL buffer, where events are organized (see Figure 7.1). On the one hand, the dedicated GPU scheduler is the responsible of mapping each entry to an asynchronous and non-blocking GPGPU calls. On the other hand, the dedicated CPU-scheduler will make use of multi-processing/threading technology (e.g. openMP). Upon the execution of an event, a new event might be generated (similar to producer-consumer processes) and pushed into the dedicated FEL. To dynamically adjust the event flow, each target sends feedback information about the instantaneous load and the execution time per event.

7.2.2 Scheduling Algorithms

In this part, we will develop the three different algorithms of the H-scheduler, namely the advanced algorithm, the rapid algorithm and the hybrid algorithm.

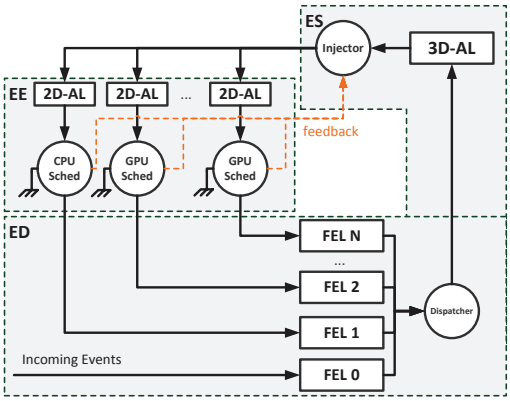


Figure 7.2: Event scheduler model

7.2.2.1 Advanced Algorithm

The advanced algorithm aims at thoroughly selecting the most adequate computing target for each event. The event flow starts from the FEL where all events are firstly inserted. The dispatcher is the first process which handles the events and as a result, the event will be pushed in the adequate position in the 3D-AL. Based on the event timestamps, the dispatcher determines the correct sub-list where the event must be inserted. Then, it starts resolving its dependency using the event descriptor. If the dispatcher detects a dependency between two events, it has three choices: (1) splitting the interval into two new ones, which of each includes independent events while respecting the timestamps correctness, (2) creating a merged event including both events or, (3) transforming the sub-list to a sequential one. Algo 1 presents the pseudo code of the dispatcher. There are two tasks related to that process: the events dependency detection and the conflict resolution. To detect the dependency of two events, the dispatcher relies on their descriptors as follows: first it explores their explicit dependency descriptors. Secondly, it computes their durations, thus if both events are concurrent then they can be considered as independent. Thirdly, if both events do not use the same data, the dispatcher concludes that they are independent (this feature must be explicitly enabled by the user). To resolve the dependency, the most adequate choice is to split the interval. However, if such a procedure induces another conflicting situation, the dispatcher creates a merged event. Finally, if dealing with the interval becomes complicated due to the large number of included events, then the sub-list will be transformed into a sorted list.

The injector processes the AL sequentially, sub-list by sub-list. If the sub-list is sequential, then all the events will be forwarded to one target. Otherwise, the injector considers each event individually based on the following routine: if the event is mono-compliant (CPU or GPU), then it is switched to the adequate sub-scheduler. If there are several instances of the target (several GPUs or CPUs) then the entry in question will be switched to the target having the lowest load currently. Concerning

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

Algorithm 1: Pseudo code of the dispatcher.

grouped entries, the injector analyzes the parallelism information to determine the target. If there are few parallel events, then the CPU is the most adequate target, and if that number is extremely large, the chosen target is the GPU. The boundaries of this decision are a function of the number of CPU & GPU cores. At the beginning of the simulation we use two arbitrary intervals where the decision is deterministic: $[1, 2{*}core{*}3{+}1] \Rightarrow$ CPU and $[200, +\infty] \Rightarrow$ GPU. Nevertheless, if the parallelism size is intermediate (number of instances greater than CPU's interval's upper boundary and lower than GPU's interval's lower boundary), then the injector inspects the load of each target and chooses the available one. Finally, it ensures the synchronization of all secondary ALs using a synchronization checkpoint at the end of each interval.

The CPU-scheduler ensures event execution over available CPU cores. At the initialization phase, the CPU-scheduler starts by discovering available resources (asking the hardware and reading the configuration file); then it creates as many execution threads as available cores. Afterward, it feeds execution threads with events as soon as possible, without dependency control. Therefore, events of one sub-list are expected to be executed in parallel over available resources. Since the H-scheduler respects a conservative scheduling approach, it does not execute events of different sub-lists concurrently. The CPU-scheduler notifies the injector once an interval has finished, and waits for its permission to consider the next one. The GPU-scheduler is slightly different, since it relies on a hybrid software-hardware scheduling mechanism. At the software level, sub-list always includes grouped entries that it translates to a CUDA call with predefined generic parameters. The CUDA driver ensures next steps, including generating threads and sending them to the GPU. At the hardware level, the embedded GPU GigaThread scheduler first distributes event thread blocks to various SMs, and then assigns each individual thread to an SP inside the corresponding SM.

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

Algorithm 2: Pseudo code of the injector.

7.2.2.2 Rapid Algorithm

The rapid algorithm is a simplified version of the advanced one which aims at minimizing the decision cost. It particularly concerns the dispatcher and the injector. It relies on a major simplification of the bottleneck of each process using a deterministic model. As for the dispatcher, the most expensive routine is the dependency resolution; in particular, splitting an interval into two independent ones requires an expensive modification of the main AL structure. The rapid algorithm applies events merging if the dispatcher detects any dependency within one interval. However, we note that the dependency detection routine can not be simplified since it affects the correctness of the simulation. Concerning the injector, the most expensive routine is the identification of the most suitable target. The rapid algorithm reduces the complexity by extending the borders of decision intervals for both CPU and GPU: the CPU interval becomes $[1, N]$, and the GPU one becomes $[N + 1, +\infty]$, where N is a tuning parameter. The second critical routine is how to determine the most adequate target if we have multiple instances (multiple CPUs or GPUs). In that case, rather than evaluating resource loads, the injector uses a predefined assignment such as a round robin mechanism which aims at ensuring a minimal load balancing based on the number of events.

7.2.2.3 Hybrid Algorithm

The hybrid algorithm aims at ensuring the maximal stability for the system. It relies on two mechanisms: the algorithms switching and the parameter recalibration. The switching mechanism changes the operating algorithm for both injector and dispatcher processes based on a bottleneck detection approach. Each process has one in-buffer which includes status flags indicating if the buffer is empty, almost-empty, almost-full or full. The in-buffer of the injector is the main AL, and that of the dispatcher is the FEL.
For each process, if the in-buffer is full, the hybrid mechanism assumes that the consumer process is the bottleneck and acts as follows: if the filling rate is between empty (E) and almost empty (AE) then the selected algorithm is the advanced. If that rate is between AE and almost full (AF) then the selected algorithm remains the advanced one, but the recalibration frequency is increased. Finally, if the in-buffer filling rate is between AF and full (F) then the selected algorithm is the rapid one. The switching decision occurs between two intervals and cannot be achieved during the execution of an interval.
The recalibration mechanism computes continuously the values of three parameters which define the behavior of both algorithms: the NCPU & NGPU of the advanced algorithm and the N of the rapid one. It maintains a statistical table which includes the average execution time of event sets in different targets. Figure 7.3 presents an example of a statistical table, where the recalibration process can assert that $NCPU = 24$, $NGPU = 192$ and $N = 48$. To compute the average execution time of a specific event set size, the recalibration mechanism considers the last M samples, which is a tuning parameter defined by the user. Therefore, the statistical

table copes with the evolution of the simulation on one hand and the hardware characteristic on the other hand.

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

Figure 7.3: Statistical table used for recalibration

7.3 Performance Evaluation

To implement the H-scheduler, we rely on cunetsim framework [16, 17]. The implementation is based on five parallelization frameworks, namely CUDA, OpenMP, MPI, the thrust data management API [12], and the PGI development Kit [146], which are briefly explained below.

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

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

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

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

5. The PGI suite is a commercial C/C++ compiler which provides several automatic and semi-automatic parallelization features. Furthermore, it incorporates a full CUDA CC++ compiler for targeting X64 CPUs. What is even

more important, is that it introduces the unified binary technology (PUB), which consists in creating a multi-target binary (GPU, INTEL CPU and AMD CPU) from an initial native CUDA code.

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

7.3.1 Scenario & Setup

We propose a large scale network experimentation scenario where we customize the benchmark methodology proposed in [16] by defining a static network topology composed of three independent activity areas (AA). Each of these follows a grid configuration where the edge of an AA contains 750 nodes as illustrated in Figure 7.4; thus each AA includes $562.5K$ nodes[1]. The scenario includes one traffic source which generates 600 uniform $128 - byte$ packets with 1 second of inter-departure time. All nodes forward unseen packets after a one-second delay to model the network latency whilst medium's reliability is reflected using dropping probability. Depending on the latter, each node decides whether or not to relay a received packet. The drop probability (DP) is the parameter which allows us to introduce a random factor on the network behavior. To provide a valuable event rate while keeping a significant variation on the number of exchanged messages we use a DP of 0.1. The simulation duration is 5602 seconds which ensures that the last generated message can reach the destination.
In addition, we introduce a second scenario where we define for each node a random inter-departure-time within the interval $[0.1, 2]$ and a DP within $[0, 0.28]$. This scenario is used to study the robustness of the H-scheduler when the timestamp of different events present a significant entropy.
Although these scenarios are outlying real networks and include major simplifications, we claim that they provide an important events load with a large rate variation as shown in Figure 7.8. Moreover, they are based on a pool of simple events which does not require powerful computing resources individually, and use a simple implementation of both nodes and channels. This approach guarantees a fair comparison and focus on the simulator performance rather than on models efficiency. Therefore, the main difference between considered approaches remains the scheduler efficiency. We use the Cunetsim framework, except for the NS3 case; thus all events are dual compliant with both CPU and GPU targets when we aim at using the GPU. As

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

for the experimental context, the used frameworks are CUDA 5.0 and Open-MPI 1.4.1. The OS is Ubuntu Linux 11.10, the PGI compiler version is the 12.9 and the Nvidia driver version is 295.41. The hardware platform is one PC including an INTEL $i73930k$ CPU (6 cores with hyper threading), 32 GB of DDR3 and three GeForce $GTX860\ 2GB$ (1536 cores for GPGPU computing).

7.3.2 Comparative Evaluation

This section aims at highlighting the efficiency of the major conservative scheduling approaches which differ in their execution targets and parallelism techniques. We distinguish three groups according to their execution targets namely CPU, GPU and CPU+GPU as summarized in table 7.1. We will use Cunetsim framework where we will change solely the scheduler except for the $ns-3$ simulator where we use the default scheduler. Moreover, we will use the same implementation of nodes and channels to guarantee a fair comparison. Based on the default scenario, we consider three metrics: the speedup with respect to a reference sequential execution, the hardware usage rate and the scheduling cost.

Table 7.1: List of different scheduling approaches

Scheduling approach	Target	Parallelism	Example
1	CPU	non	sequential
2	CPU	Op	NS3 scheduler [105]
3	CPU	Msv + Op	Cunetsim-CPU
4	GPU	Op	[100]
5	GPU	Msv	Cunetsim
6	GPU	Msv + distributed	Cunetsim-GPU
7	GPU+CPU	Msv + Op	H-scheduler

Op =opportunistic and/or optimistic, Msv= Massive

Figure 7.5 shows the normalized speedup of each approach with respect to the reference sequential runtime obtained as the average runtime of the sequential $ns-3$. We observe that GPU-based approaches are extremely faster than all CPU-based ones. However, we observe that the opportunistic approach on GPU (case 4) which presents a speedup of $(40x)$, does not consider the SIMD architecture of GPUs. In fact, its results are due to the efficiency of the hierarchical GPU memory and the existence of 24 independent SMX[2] in the used platform. On the other side, the GPU-based approach which relies on massive parallelism concepts (cases $5-6$) presents an outstanding speedup which varies between $400x$ and $900x$. Nevertheless, we note that D-cunetsim cannot reach the maximal expected speedup $(3*400x)$ while it uses 3 GPUs. According to a detailed profiling analyzes, we can assert that the scheduler is the bottleneck of the D-cunetsim. In addition, we notice that

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

the H-scheduler is twice faster than the default scheduler while both use the same hardware. This considerable gain demonstrates that the hybrid scheduling approach reduces significantly the bottleneck impact.
Figure 7.6 presents the average hardware usage rate of each approach for the CPU and GPU. It reflects the used resources to ensure the simulation. As expected, CPU-based schedulers ignore the GPU, however their CPU usage differs. The sequential scheduler uses on average 20% of the CPU which represents the usage of one CPU core. $Ns-3$ uses on average 80% of the CPU. Since there are 6 instances of $ns-3$, each of which uses one CPU core, we can deduce that the corresponding waste of resources (about 20%) is due to the communication overhead. Therefore, we can infer that the event grouping policy presents a significant *added value*. The fourth case shows a mitigated score; it uses a small fraction of CPU and GPU resources but outperforms all CPU-based schedulers. This behavior confirms that GPU provides signficant gain but that the software design must be reconsidered to cope with hardware specifications.
Regarding dedicated GPU schedulers $(5-7)$, we notice that the mono-GPU version of Cunetsim (case 5) efficiently uses the GPU and one core of the CPU; that it gives the expected results. On the other side, we note that the H-scheduler reaches the maximal CPU and GPU usage rate. In particular, it outperforms the distributed scheduler which does not exceed 80% of the GPU usage rate while the H-scheduler achieves almost the 100%. These observations prove that the H-scheduler is able to maximize the usage of powerful solutions.
Figure 7.7 presents the average CPU usage rate of the scheduling process regardless of the simulation. Unsurprisingly, the H-scheduler needs on average between 6x and 8x more resources than CPU-based approaches and up to 3x more than GPU ones. In fact, it uses at least 4 different threads to achieve the scheduling process. In the experiment case, we use 6 threads distributed as follows: dispatcher, injector, one CPUs-cheduler and three GPU schedulers. Moreover, these threads work in parallel since the different data structures ensure the role of intermediate buffers.
We conclude that the H-scheduler is able to maximize the simulation efficiency, compared to a classical one. Moreover, it is able to deal natively with heterogeneous platforms if events are compliant. However, this efficiency requires additional dedicated resources compared to centralized approaches.

7.3.3 Performance Analysis

The H-scheduler is composed of several processes, each of which has two algorithms with the ability to switch between them based on the feedback mechanism. In addition, each algorithm reconfigures itself periodically. In this section we propose to analyze the impact of each algorithm on the global behavior of the H-scheduler. To study the impact of each algorithm, we conduct the following experimentation series: first we will measure the average event rate per simulated timestamp, generated during the simulation across 100 runs. We realize so many runs as the shape of the curve of Figure 7.8 appears so perfect for a simulation including a random

factor. Nevertheless, we verify that the message propagation on the proposed network respects that shape. In particular, each peak of this curve reflects a maximal network activity in one AA. The second scenario which includes randomly variable connections between nodes has a less regular curve[3].

Using the default scenario we proceed as follows: first, we only enable the advanced algorithm, secondly we enable the rapid one, and finally the hybrid algorithm without any reconfiguration process or setting a timer, and finally we consider the H-scheduler as described in section 7.2. The simulated time is 5602 seconds while the execution time varies between 413 and 670 seconds which complicates the comparison. To present an understandable representation, we normalize the output event rate with respect to the following formula $Outputrate = Nevents/Executionduration$. Corresponding results are shown in Figure 7.9.

First of all, we notice that the advanced algorithm (red curve) is in general more efficient than the rapid one (green curve). However, the latter punctually achieves higher output rate in some cases, especially at the end of the simulation, characterized by a reduced number of messages. Furthermore, we notice that the hybrid algorithm provides better results while it introduces a large variability during time as shown by the width of the curve (blue). We can assert that, when allowing each component to use the most adequate algorithm as a function of the situation, we reach a higher event rate with a risk of inefficient oscillation. Finally, the full H-scheduler which involves both hybrid algorithm and the continuous recalibration presents unquestionably the highest output rate but also the most variable behavior. In fact, we can distinguish four distinct and quasi-parallel sub-curves, each of which presents the maximal achievable rate of one of the available computing processors(1 CPU and 3 GPUs). We note that the recalibration procedure allows the scheduler to match rapidly the typical hardware parameters which gives a significant gain of almost 30% compared to the switching mechanism alone and about 90% compared to the default scheduler.

To illustrate the impact of the continuous re-calibration, we propose to analyze the evolution of two parameters during the simulation: first we will consider the average decision path length, computed as the average number of steps required to make the decision on where the event will be directed. Secondly, we will consider the average scheduling interval length during one simulated second. Concerning the path length illustrated in Figure 7.10, we identify a first phase, where the length seems extremely variable, oscillating between 1 step (18%) and 5 steps (13 %). This represents the learning phase, which was arbitrary fixed to 500 simulated seconds. Afterwards, we observe a transition phase where the average decision path length decreases rapidly until reaching a steady state where the scheduling decision needs in average between one and two steps. We conclude that the continuous recalibration allows a significant gain in term of scheduling cost without compromising the decision quality. Regarding the scheduling interval length illustrated in Figure 7.11, we notice that

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

the interval length is sensitive to the experimentation conditions; therefore its value decreases to 1-5 ms when the number of messages is high and increases to 10-15 ms when events are mixed. Considering that message events are natively dependent, this behavior reflects the events relationship in firm way. Moreover, we notice that there is no learning phase since the unique rule during the whole simulation is to maximize event parallelism.
Accordingly, we can conclude that the H-scheduler presents an interesting ability of dealing with variable simulation rates under large scale conditions while maximizing the hardware usage rate even in a heterogeneous context. Moreover, the re-calibration procedure and the dynamic behavior allow a significant support of the hardware characteristics without prerequisite knowledge.

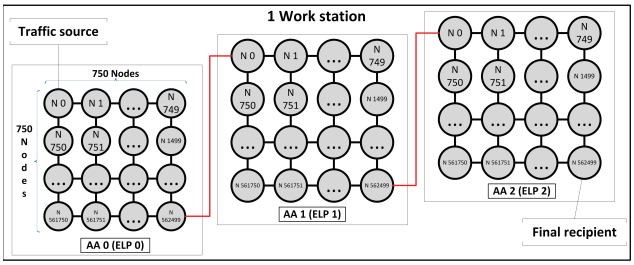


Figure 7.4: Topology of the benchmarking scenario

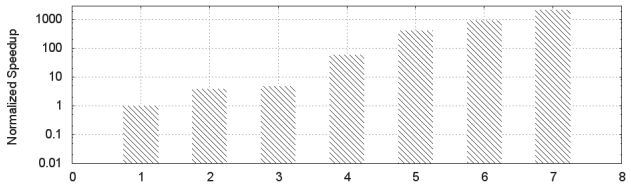


Figure 7.5: Normalized speedup with respect to the sequential runtime

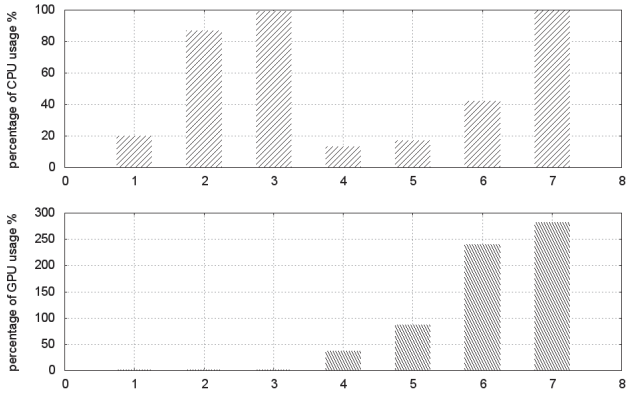


Figure 7.6: The hardware usage rate.

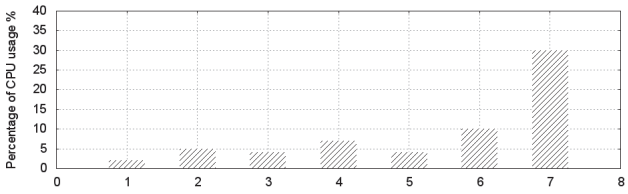


Figure 7.7: The scheduling cost.

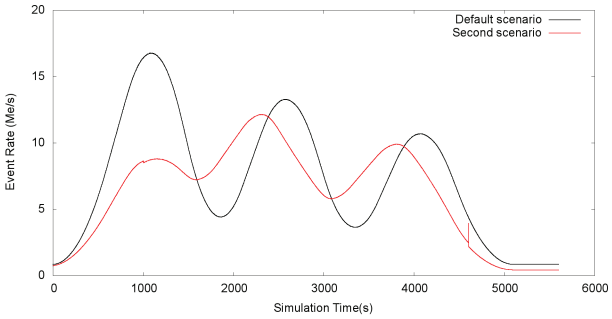


Figure 7.8: Variation of the input rate vs Time

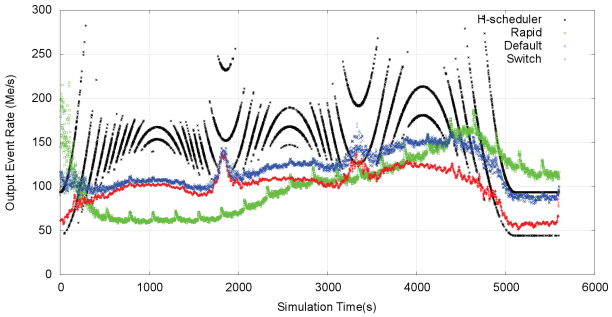


Figure 7.9: Output event rate of different algorithms.

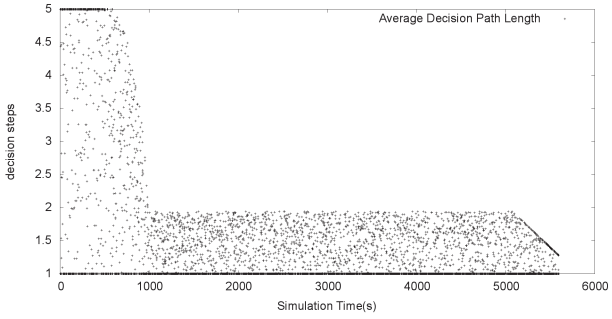


Figure 7.10: Average decision path length during the simulation.

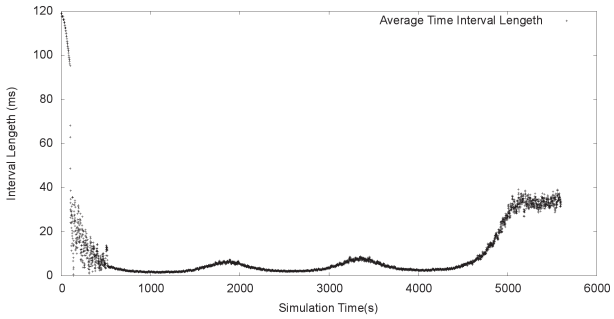


Figure 7.11: Average interval length during the simulation. It closely reflects the events dependency.

7.4 Related Work

Improving the efficiency and scalability of DES remains a challenging issue for modern modeling approaches that require complex and sophisticated representation. In such context, PDES is commonly used as a scalable and efficient solution when compared with sequential approaches [33]. PDES relies on the partitioning of the model over several logical processes (LP)s collaborating with each other to perform the whole simulation [111, 62]. However, respecting the simulation correctness while dealing with parallel execution makes event management extremely expensive. This is also acknowledged as one of the critical limitations of large parallel simulations, especially when dealing with heterogeneous resources, and raises two issues: data representation and event scheduling [110].

To store future events, most of the PDES frameworks use a sorted data structure. In the literature reviews, the efficiency of central data structures was largely studied for both sequential and parallel execution, e.g. central event list (CEL) [122]. Nevertheless, under large parallelism conditions, such a data structure becomes the bottleneck. Authors in [33] address the efficiency of three CEL implementations, namely the heap, the splay tree and the calendar, and they conclude that the performance of the CEL concept remains mitigated when thousands of concurrent processes access that structure. Therefore, the CEL implementation needs to be parallelized to cope with the parallel architecture of heterogeneous computing resources. The concurrent priority queue [134, 38] is a relevant solution to access and manage the CEL in parallel. An event list or message queue is usually distributed to each logical process in a PDES with its own local clock. The concurrent insertion and deletion of the priority queue, by involving mutual exclusion or atomic functions, leads to the improvement of the overall performance using a global event list [122]. In the same sense, Chen *et al.* propose a distributed queue which considers multi-core CPUs [27]. However, the above mentioned mechanisms, mutual exclusion and concurrent priority queue, are target-dependent, they could not be directly applied to GPU targets. A different point of view was proposed by Park *et al.* [100], which relies on a hybrid time-event driven simulation based on a GPU-oriented CEL con-

cept that uses a linked list implementation. Despite the fact that this approach has been developed with the aim of improving the GPU-based simulation, the overhead of managing a large number of parallel event remains an open issue due to a limited number of concurrent access to the same physical memory.
In the DES and PDES, a large portion of the overall simulation time is used for the event scheduling [102]. Moreover, the efficiency of the scheduler also depends on the synchronization method [84]. The conservative approaches prohibit out-of-order event execution, which in most cases is based on the lookahead concept to preserve the causality rule [46]. Parallel event scheduling is separately studied for a multi-core CPU target [139, 36] and GPUs [100]. Both approaches use a central event queue and several independent threads to fetch the next event from the queue. The multi-threading approach is also applied to reduce the scheduling cost and increase the simulation efficiency but only for a limited number of cores (4 and 8 respectively). Authors in [99] propose a dedicated GPU scheduler based on the SIMD programming model where the event queue is splitted into several sub-queues to avoid central bottlenecks. All the above mentioned approaches regard a GPU core as a CPU core, which in turn reduces the achievable gain.
The optimistic approach allows an out-of-order execution while ensuring the simulation correctness. Several optimizations were introduced recently to increase the efficiency of optimistic parallel simulations over multi-core CPUs while keeping a reasonable backward compatibility with a standard software architecture [138, 88, 27]. Although such an approach increases the general efficiency by relaxing a substantial limitation of the conservative approach, it introduces a significant memory overhead related to the state-vector saving mechanism, which becomes critical when targeting a very large logical processes.
In contrast with previous works, the H-scheduler design consider extremely large LPs while maximizing the simulation efficiency over heterogeneous computing architecture. In particular, it outperforms prior works by considering events as flow and detecting the system bottleneck on the fly in order to adjust the behavior of the scheduler dynamically.

7.5 Discussion

This chapter aims at emphasizing the importance of using heterogeneous computing resources to maximize the simulation efficiency and scalability. Thereby, we use an adequate machine including identical GPUs and a well-designed scenario, suitable for a high events rate. Nevertheless, we identify five issues which need to be discussed: first, we do not consider the data structure filling issue. In fact, all events have the same importance whatever the status of the concerned buffer. An adaptive policy may use a RED [41] approach to manage different buffers. Secondly, the management of the algorithms switching relies on a loopback mechanism which measures the load of each auxiliary resource. While this approach seems efficient to cope with very large scale simulations, analytical evaluation demonstrates that it

may cause unsuitability when the simulation pattern becomes unpredictable (such as the second scenario). Therefore, a weighted flow management appears as a solution to maintain the queue stability.
Thirdly, we use a limited range of computing hardware, including one GPU family and one Hexa-core CPU. We expect to evaluate the robustness of the H-scheduler under large heterogeneous conditions where we combine powerful CPUs (Xeon E5 2650) with professional GPUs and accelerators (Xeon Phi).
In addition, we notice that switching between the scheduling algorithms and execution targets may induce an instability in event rate. Thus we apply a timer to stabilize the system. We expect that a smoothed transition between algorithm may increase the efficiency while providing enhanced stability. To conclude this discussion, we highlight that we rely on a data-abstraction mechanism which allows any event to access any data whatever its location. Therefore, the H-scheduler omits the data-access cost when computing the most suitable execution target. Nevertheless, we note that the data locality is an emerging issue which may be considered on the event scheduling level, especially when targeting heterogeneous computers.

7.6 Conclusion

Discrete event simulation is widely recognized as an essential tool to analyze complex systems. Modern structures often require sophisticated models while simulating a large number of entities in continuous interaction. However, the scalability of this simulation remains challenging, whether in terms of runtime as well as the number of simulated entities. In this context, parallel (and distributed) discrete event simulation is actually the most relevant solution allowing to significantly increase the scalability. However, event scheduling during the simulation over PDES requires optimized design to deal with emerging hardware innovation while respecting the simulation correctness. In particular, new computers are transformed into a heap of heterogeneous processors, each of which is more adequate for a specific task. Nonetheless, most of the existing schedulers are derived from a sequential concept which reduces their ability to cope with parallel hardware specifications. Accordingly, the events flow is generally under expectation, and the idle time is consequent.
We present in this work a new scheduling approach which aims at maximizing the event throughput over heterogeneous computers. According to these considerations, we rethought the event-driven simulation architecture to consider event flows rather than individual events. Therefore, events are clustered as a function of their process and timestamps for the scheduling step while events flows are further directed to the adequate execution target. The implementation of this concept is denoted as the H-scheduler addressing the heterogeneous computing. It considers both GPU & multi-core CPU. In the current version, the H-scheduler is composed of four components: the dispatcher which computes the possible parallelization issues for events, the injector which determines the execution target of each parallel event group, the CPU-scheduler which ensures the execution of events on the corresponding CPU and

the GPU-scheduler which ensures the execution of events on the corresponding GPU. Both injector and dispatcher present an abstraction layer which simplifies the user' job. Besides, both CPU-scheduler and GPU-scheduler are achieved according to a hardware/software co-design methodology. Therefore, we combine the simplicity of usage and efficiency of specific target-oriented solutions.
Accordingly, experimental results demonstrate that the H-scheduler overcomes the majority of existing schedulers. In particular, it is able to achieve a reference simulation 1200x faster than the sequential one and 2x faster than the original Cunetsim scheduler while using the same workstation.

Chapter 8

General Purpose Coordinator-Master-Worker Model

8.1 Introduction

In distributed simulation, it is fundamental to rely on a dedicated software design that copes with the scale. In fact, if the software design relies on a classic master-worker approach, increasing the size of the simulated system leads to a nonlinear increase in the required resources and the execution time, which in turn reduces the simulation efficiency [48].
To speedup a large scale simulation, there are two trivial approaches: 1) parallelism and/or distribution of the simulation over several LPs. 2) usage of 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. In particular, the simulation overhead relative to correctness mechanisms remains significant. Indeed, the expected gain is asymptotic and cannot exceed a given limit that depends on both the software and the hardware inherent characteristics [47]. In the literature reviews, the initial software architecture used to define a distrusted and parallel simulation was the flat architecture where LPs collaborate to realize the simulation using distributed algorithms for communication and synchronization. Such a design is widely used for small to medium scales (in terms of LPs number). However, its *relative overhead* increases rapidly according to the simulation scale. To limit such an overhead when targeting large scale simulations, a two level hierarchical architecture was introduced, where a specific process (also known as the server) ensures the management of the simulation. The involvement of this process varies from one implementation to another.
The master-worker (MW) model is an example of a two-level hierarchical architecture that handles efficiently meta-computing systems [98]. Such a design is optimized for recent public hardware; however, specific considerations must be done to increase (1) the data locality as flops are cheap, but communication is expensive, and (2) 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, hierarchical approach was proposed in [4]. Authors propose a new concept based on the interaction between CPU-based and GPU-based components. In addition, a specific consideration for the data local-

ity was introduced. Nevertheless, this approach does not address the GPU memory restriction and as a consequence, a constant synchronization delay [26].
In contrast with existing master-worker software architecture, we had proposed [17] an enhanced design that introduces a coordination process on the top of the system. The proposed model is donated as the CMW and aims to provide an asynchronous hierarchical architecture, in which each master manages a local group of workers within the same addressing space. The set master-workers defines a logical process which is able to handle a large part of the simulated population. The head process, on the other side, manages the simulation through a distributed environment. Despite the advantage of the CMW model that we highlighted in chapter 7 and in [17], it presents three major issues:

1. The software design is close to the hardware which reduces its portability.

2. The model presents some stability weakness under very high load.

3. The absence of a priority management mechanism maximizes the instability.

It has to be mentioned that we denoted the initial version in which relies the hybrid scheduler introduced in chapter 7, as the basic CMW in order to differentiate it from the GP-CMW.
In this chapter, we aim to overcome these limitations and to consider meta-computing systems for greater flexibility. The main rule is to maximize the interactions inside a computing node with minimum communication overhead outside. A specific consideration is also given to the stability and extensibility of the solution in order to support future hardware trends. Thus, we propose a general purpose version of the coordinator master worker model (GP-CMW) that introduce two abstraction layers: the priority management layer (PAL) that maximize the stability by separating both control and data plane; and the hardware abstraction layer (HAL) that regroups features thoroughly bound to hardware such as heterogeneous event scheduling and internal communication mapping. The GP-CMW is an asynchronous hierarchical model: at the top level, the coordinator ensures the global time synchronization and the load balancing among the masters. On a 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 notified as an extended logical process (ELP). The PAL is an intermediate layer between the coordinator and masters on the one hand, and different masters, on the other hand. The HAL is a local layer in each addressing space and includes the heterogeneous events scheduler that manages event execution through all available resources. The asynchronous feature is ensured by the fact that the synchronization is carried out at two different scales: the master-coordinator level uses a large granularity synchronization while the master-worker level uses a fine granularity.

The reminder of the chapter is organized as follows. In Section 2, we summarize the related works. In Section 3, we present the GP-CMW model and its features. Section 4 presents the benchmarking scenarios and validation results for the GP-CMW model in comparison with the basic CMW model. Finally Section 5 concludes the chapter and provides a short contributions.

8.2 The General Purpose Coordinator-Master-Worker Model

The GP-CMW model is a software architecture that considers distributed large scale simulation across heterogeneous meta-computing resources. It is designed around three software components and two optimization layers:

1. **C** *the coordinator:* is a top-level CPU process with two essential tasks: load balancing and synchronization among all the active masters.
2. **M** *the master:* is a CPU process representing an intermediate entity. 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.
3. **W** *the worker:* is the elementary actor of the GP-CMW that performs the simulation routines and interacts with the input and output data.
4. **HAL** *the hardware abstraction layer:* is a unique process per addressing space that performs event scheduling through workers. In addition, the HAL maximizes the usages of specific communication resources into the same addressing space.
5. **PAL** *the priority abstraction layer :* manages the communication priority between the coordinator and masters as well as between masters. It gives the highest priority to managing flows in order to maintain the system stability.

In the GP-CMW model, the simulation is first distributed over a certain number of workers for the considered simulation scenario. Then, workers are partitioned into separate simulation instance according to the user-defined spatial and/or operating policies [67, 119]. Each simulation instance is managed by one master, and all workers interact with the outside world uniquely through the master. In a typical case, each simulated instance is performed by one master on one computing nodes. Figure 8.1 summarizes the hierarchical architecture of the GP-CMW model where we highlight the separation between simulation and control plans. In particular, we observe the role of the HAL to extend the execution context of one ELP over available resources (CPU and GPUs).
In the following sections we will detail the event management model which emphasizes the events-flow concept. Secondly, we will highlight the particularity of the

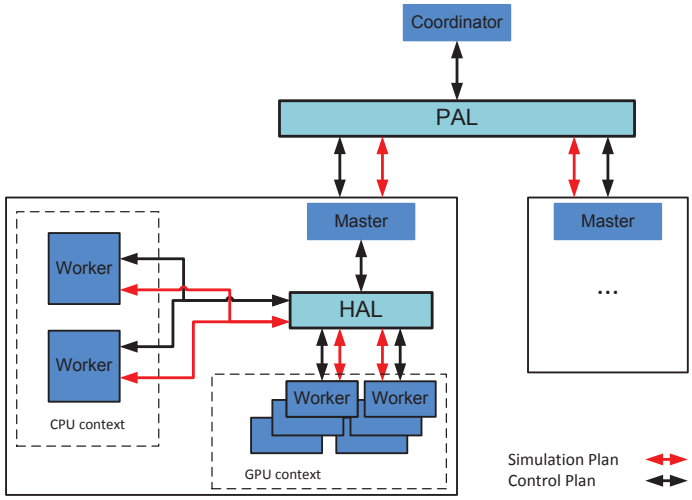


Figure 8.1: The GP-CMW blocks diagram:

The coordinator flows goes exclusively through the PAL. the PAL manages also master-master communication. Inside a given addressing space the HAL manages the interaction between the local master and involved workers. It includes both events scheduling and communication switching.

optimized synchronization model and finally we will discuss the hierarchical communication model.

8.2.1 Events Management: Description, Scheduling and Execution

In DES an event represents the execution of one state, activity, model or algorithm with concrete timestamps and 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 8.2(a)). By agreement, there are two types of generated events: that defined by the user scenario (i.e. via a permanent events' generator) and that generated recursively due to a previous event execution. Whatever the generation method, an event has specific timestamps that define when it must be executed with respect to the simulation time. A sequential scheduling process consists of sorting incoming events based on their timestamps and buffering them into a FIFO list in order to be executed. A basic parallel scheduling process aims at executing events in parallel if they have exactly the same timestamps (called the conservative approach) . Optimized scheduling models relaxed this constrain using a time window to determine parallelizable events (opportunistic approach) [43, 131, 114]. Advanced parallel scheduling algorithms analyze events dependency to maximize the parallelism rate while conserving the simulation correctness [144]. In case of parallel scheduling for

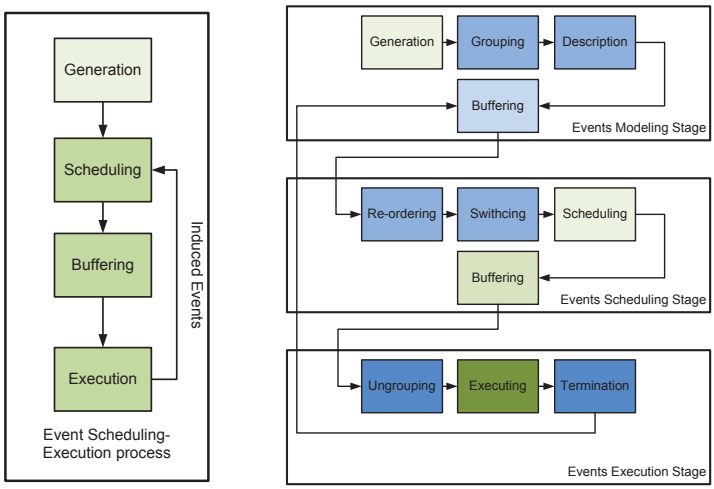
multiple execution targets, a central scheduler becomes an imminent bottleneck. In fact, such a situation can be modeled as a workflow with one producer and many consumers [1:N], where consumers are the execution processors, and the producer is the scheduler. Therefore, increasing the number of consumers with direct exchange order results inevitably on 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 on the one hand and the number of execution processors on the other hand. Thereby, the majority of parallel schedulers deal with a limited number of simulated elements (thousands per LP) and execution processors (Typically 6-64 in the current state of the art [28, 90]).
In contrast with traditional approaches, the GP-CMW events management proposes three features:

1. The separation between the event description, scheduling and execution.
2. The concept of *flow of events* .
3. The grouping/ungrouping of events representation during the simulation.

The event description relies on an enhanced representation that includes common information (id, timestamps, callback) but also dependency meta-data that 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 description. Indeed, the events management process on the GP-CMW separates the event description from the scheduling and the scheduling from the execution. Thus, the description is achieved by the master while the scheduling is done by the HLA and the execution by a workers pool. The GP-CMW model combines the generation of cloned independent events (CIE) with the detection of foreign independent events (FIE) that can be potentially executed in parallel to maximize the simulation efficiency. Finally to bypass any potential bottleneck due to an increasing events rate under large scale simulation conditions, the GP-CMW model represents the CIE as one entry. This simple yet efficient technique provides a significant gain in terms of scheduling cost. To realize this approach, we propose to modify the event life cycle by breaking it down into three independent stages: the events description, the events scheduling and the events execution (see Figure 8.2).
According to these modifications, we can expect that we combine the advantage of an efficient hardware usage with a reasonable abstraction level on the software design. Moreover, the usage of several intermediates buffers and processes guarantees that the flow of events becomes smoother, reducing the instability of the system during the execution.

8.2.1.1 Events Description Stage

On the events description stage, the master pre-processes incoming and generated events in order to reduce the complexity of scheduling stage. Thereby, it includes



(a) The default event life cycle in the DES includes four mandatory steps

(b) The separation between the event modeling the event execution provides more flexibility and reduces the impact of potential bottlenecks.

Figure 8.2: Events Life Cycle

four steps:

1. The generation: 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[1].

2. The grouping:the master identifies and groups CIE into one entry. The main issue of the grouping step is to preserve the event descriptor so that it can be reversible. Thus, the simplicity of events parameter grouping determines the efficiency of the procedure.

3. The enhanced event descriptor (EED):the master generates and extends the event descriptor to provide additional information for event groupings. In particular, it includes: *dependency information, execution timestamp, I/O data access, structure information and execution targets*

4. The buffering: To conclude this stage, events descriptors are stored into an intermediate buffer, where they still waiting to be scheduled and further executed.

It has to be mentioned that the main issue of the grouping step is to preserve the event descriptor so that it can be reversible. Thus, the simplicity of events parameter grouping determines the efficiency of the procedure. Figure 8.3 presents

[1] That events can be recursively generated by other events during the execution or received from outside.

two grouping examples, case A is non-optimized and generates additional work while the case B is optimized and simplifies both grouping and ungrouping steps. Indeed, the main part of work remains in charge of the system designer which must provide identical parameters for CIE.
The five components of the enhanced event descriptor are described as follow:

1. Event dependency information: defines if an event has one or multiple dependencies (needs current output as input) that fall within the same time interval [13].

2. 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.

3. I/O data access: defines permissions given to an event to read and/or write a shared memory area.

4. Event structure information: defines the type of the event (CIE or isolated). In case of CIE, it includes multiplicity information.

5. Execution targets: defines where an event could be executed, CPU, GPU or both.

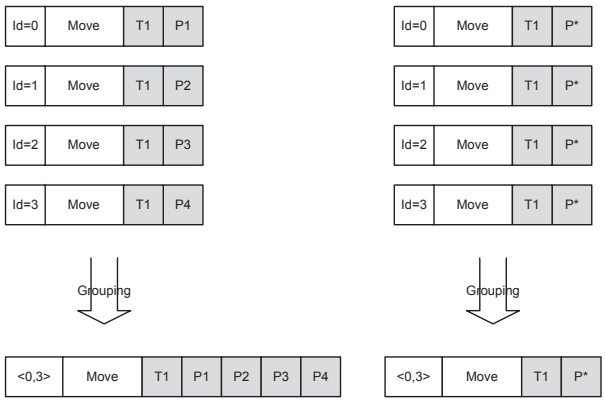


Figure 8.3: Examples of events grouping:

This figure presents two events grouping cases: If the events that will be grouped have different parameters, the grouping process must concatenate all of these parameters which complicates the operation. However, if all events share the same pointer to a common structure (table, vector, flow...) then the grouping process generates the grouped entry rapidly.

8.2.1.2 Events Scheduling Stage

The event scheduling stage of the GP-CMW model is the achievement of the scheduling expertise that we acquired during this thesis. It is derived from the H-scheduler presented in chapter 7. However, the scheduling approach of the GP-CMW is more generic and consider extreme situation in order to increase the global stability. From a conceptual point of view, it includes four steps: the reordering, the switching, the scheduling and the buffering.

1. **Reordering:** Received Events are reordered and grouped according to time-windows. This pre-scheduling step reduces the complexity of the following steps. The output of this step is a group of events that can be executed in parallel.

2. **Switching:** Pre-scheduled events will be switched to one of the available computing resources. In particular, grouped events (CIE) will be switched to the GPU and isolated one will be switched to the CPU.

3. **Scheduling:** The *scheduling* concerns each computing resource independently. An associated sub-scheduler manages the given events. However, these sub-schedulers are synchronized at the end of each time window.

4. **Buffering:** Scheduled events are buffered in order to be executed. This step is fundamental because each sub-scheduler may have several executing resources which must be supplied from a common buffer.

8.2.1.3 Events Execution Stage

The event execution stage is the concrete achievement of the simulation. Thus, its main step is the execution! However, it includes two mandatory steps: the ungrouping and the termination.
Conceptually, the ungrouping step consist on the creation of several events (or events descriptor) from one CIE entry in order to be executed. The concept did not specify how, since the ungrouping method depends on the target hardware. The execution step is achieved by a pool of threads that act in parallel. The nature of that threads is not specified on the concept. The designer is free to implement the model according to its relative constraints. The termination step represents the after-execution management operations such as memory liberation and acknowledgement, but also the handling of induced event that can be generated during the execution. The implementation that we propose relies again on the H-scheduler and requires the collaboration of the HAL and workers pool.
Implementation: on the event execution stage, all sub-schedulers act simultaneously to ensure events processing. Regarding the CPU-scheduler, the process is the following: first it creates as many threads as available cores. Afterwards, it pops events from the corresponding 2DAL and affects them to the nearest free thread. If the CPU-scheduler finds grouped events, it relies on the OpenMP API to ensure

the parallel execution. If an executed event generates a new event(s), there are two approaches: if the generated events can be executed during same time, then the sub-scheduler short-circuits the system and executes it directly. Otherwise, the new event(s) is re-injected on the simulation loop. The GPU-scheduler is slightly different since it relies on a hybrid software-hardware scheduling mechanism. On the software level, it always handles grouped entries that it translates to a CUDA call with predefined generic parameters. The CUDA driver ensures next steps, including generating threads and sending them to the GPU. At the hardware level, the embedded GPU Giga-Thread-scheduler first distributes events threads blocks to various SMs, and second assigns each individual thread to an SP inside the corresponding SM.

8.2.2 The Synchronization Mechanism of the GP-CMW Model

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. Accordingly, it defines a referential time of the 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. Regarding time advancement, there are two methods relative to the DES: the time-driven and the event-driven simulation. In time-driven method, the clock increases sequentially from one value to the next with a predefined granularity which defines a time interval. In distributed context, this 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 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 distributed context. In general purpose distributed simulation, the lookahead is the key ingredient for all conservative synchronization methods. A simplified definition is that the ability of a simulation instance (also known as a logical process in the literature reviews) to predict future behavior with respect to 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 will it insert or delete an event from the event list of the instance Q with timestamps $t > s$ [92].

In the GP-CMW model, we aim at maximizing the efficiency of the distributed simulation while minimizing the management overhead. Consequently, we opt for a hy-

brid and hierarchical synchronization mechanism where we separate the coordinator-master plane from the master-worker plane[2]. 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 synchronization time points (denoted as checkpoints) that must be acknowledged by all active masters. The elapsed time between two consecutive checkpoints represents one work unit (WU). The mechanism is an active and conservative method that have the benefit of avoiding any additional recovery routines to preserve the correctness. However, the usage of a conservative checkpoints-based mechanism implies that all ELPs must wait until receiving the release message of each finished WU. That waiting delay may become significant if the designer uses a short WU. The master clock is time driven and is incremented according to the time intervals computed by the events scheduler. In particular, each master acknowledges the HAL by the duration of the WU; yet the embedded scheduler divides this duration into several independent and contiguous intervals. The rule is that all the events of the same interval can be executed in parallel despite their timestamps divergence. It has to be mentioned that during a WU, each master can progress according to its inherent speed which may induce a correctness issue. Therefore, we apply an optimized lookahead protocol to ensure a correct communication between all ELPs [129]. Finally, the worker clock is the unique element which is event-driven. Indeed, only the clock of active workers will be updated to the current master time. An active worker is that which will execute an event during the upcoming timestamps while a passive worker is one that did not have something to do in that time. Accordingly, the worker clock jumps from the current event timestamps to the next one, similarly to a traditional event-driven simulation, from a worker point of view.

A second important feature that distinguishes the GP-CMW model is the ability to support variable WU duration. In fact, this feature allows each master to request the duration of the next WU. Therefore, the coordinator computes the optimal duration according to received requests. Different policies can be applied such as the Max, the Min and the average. Depending on the simulation nature, the user may define the most adequate one.

8.2.3 GP-CMW Communication Model

In this section, we will survey communication features of the GP-CMW model. First, we will present the hierarchical model that governs the interaction between different entities. Secondly, we will discuss three issues that influence the simulation efficiency and the resolving features that we propose to limit the damage.

[2]We have omitted to represent the PAL and HAL on the synchronization process because they have not interfered with the system logic.

8.2.3.1 Hierarchical communication Model

The GP-CMW model relies on a hierarchical communication policy between simulated entities; defined as the interaction between workers. The GP-CMW model defines four methods that aim at maximizing the usage of available communication buses:

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 destination(s) 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 destinations will receive distinct messages.

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 memory access. 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 foreign ELPs, the source worker writes the message on the out-buffer of its master, denoted as the source 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 torque (id, physical location) of each worker. Thus, when a worker requests a communication with another one, it fetches the locality information in that table in order to configure its communication method. Moreover, it caches this data for future usage.

8.2.3.2 Communication Issues and Features

In distributed and parallel discrete event simulation, the communication between different entities remains a renewable challenge. While computing power was the domain limitation in previous decades; it is nowadays the communication that counts largely match than the computing power. When we started the creation of a new parallel and distributed simulation framework, we faced three major issues: bursting, priority and redundancy. The first and second issues concern distributed context. The bursting occurs when several ELPs exchange an important number of small messages. It induces a significant overhead on the network and increases the simulation latency. The priority issue concerns the management of the simulation flows through the distributed infrastructure. It becomes critical when the global communication load reaches the hardware limit. These two issues are related to the distributed context while the redundancy issue is a parallel limitation on shared memory context. In fact, the redundancy concerns the management of duplicated messages (i.e. multicast communication where the same message is addressed to

several destinations). The most common policies are these of smart pointer and independent real copies [145]. The first minimizes the memory usage with risk of a bottleneck while the second avoids central management in spite of an excessive memory usage.
To deal with these issues, the GP-CMW model addresses each of these independently. To reduce the impact of the bursting, the GP-CMW model includes an aggregation mechanism at each master that works as follows: the master opens an internal reception buffer where it saves all the messages coming from simulated entities and going outside. It defines a small listening window where is catches and delays all outgoing messages. At the expiration of that window, it sorts this buffer and creates one aggregated message per destination. Even if the user defines a very small window, experimental results prove that this approach remains relevant as detailed in section 8.3.2. On the other side, when a master receives an aggregated message, the decomposition routine is relatively easy and still faster than switching each message independently. By agreement, we consider this operation as a main feature of the PAL.
To deal with the variable delay due to the concurrence between flows, the GP-CMW model relies on the PAL that privileges control flows. Furthermore, it gives the highest priority to the synchronization flow. This concept is not innovative in itself and was applied on different simulation and emulation frameworks [78]. Nevertheless, it is a new feature of the GP-CMW model compared with the basic one that increases significantly the stability of the system. Besides, it works transparently without any additional configuration which simplifies the user experience.
Finally, to deal with the redundancy issue, the GP-CMW model relies on *independent real copies*. Thus, if one simulated entity addresses one message to several destinations it will create as many copies as destinations; when a message crosses a network it will be rewritten as many times as it must be forwarded. While this choice implies a phenomenal increase in the memory usage, it has the advantage of avoiding any bottleneck related to the centralized processing of smart-pointers. Moreover, it is natively compliant with both GPU and CPU which makes it the best candidate for parallel and distributed simulation over heterogeneous computing resources.

8.3 Comparative Evaluation

To highlight the efficiency of the GP-CMW model, in contrast with existing software architecture, we propose two evaluations: first, we propose to simulate a large scale network scenario that involves up to one million of mobile wireless nodes and we compare three simulators: the distributed $NS-3$ and two versions of cunetsim, one uses the basic CMW architecture and the second uses the GP-CMW. By agreement, we denoted the first as basic CMW and the second GP-CMW. This experiment focuses on the macro efficiency of the proposed solution. Second, we propose to simulate a massively multi-player online game. This simulation requires an impressive computing power and cannot be done using existing simulators. Thus,

we use cunetsim, and we compare some software configurations in order to highlight the impact of each optimization.

8.3.1 Comparative Performance Evaluation

In this section, we propose to compare the performance of three distributed network simulators when targeting large scale simulation. The scenario involves up to one million of mobile wireless nodes. The experiment is achieved on the curie supercomputer using hybrid computing nodes (GPU+CPU). Since we focus on the macro performance, we highlight the simulation runtime and the synchronization delay. The evaluation parameter is the number of LP (ELP for cunetsim) that achieve the simulation.

8.3.1.1 Experimentation Scenario

The experiment scenario represents a grid of $32 * 32$ activity areas (AA). Each AA is a 3D parallelepiped of $200 * 500 * 500m$. Each one includes 1020 mobile nodes and 4 wired nodes. Mobile nodes move in the space according to a random-way-point model with an average speed between $1 - 5m/s$. Wired nodes are in corners of the AA ($Z = 2$). Each wired node is connected to some other wired nodes from other AAs as represented in figure 7.4. In the matter of the wireless features, the maximal transmission range is $100m$. The channel is modeled using a QUDG where the $Rmin = 20m$ and the $Rmax = 100$. The simulation time is fixed to 1000 seconds. A node is able to send and receive up to 10 packets per second. Each node has to send 200 packets to a given server. The decision to generate a packet relies on random number generator. Nodes retransmit unseen packets using a flooding protocol. That simplified model presents two advantages: first it guarantees the fairness of the comparison since it relies on the same models implementation. Second, it highlights the efficiency of the simulator' kernel despite the impact of model implementation efficiency.

8.3.1.2 Experimentation Setup

The goal of this evaluation is to study the robustness of the software design under large scale distribution condition. Thus, we limit the size of each AA to 1024 nodes which is an adequate size for $ns - 3$ if we associate one AA with each LP. However, this is a very small size for cunetsim that copes with larger simulation per LP (details in chapter 7). Accordingly, we achieve 6 experimentation series where we vary the number of LPs from 32 to 1024 [3]. All these experiments wer achieved on the TGCC-Curie supercomputer using its hybrid nodes. It includes 144 computing nodes, each of which includes 2 Intel westmere Xeon CPUs (8 cores) and 2 NVIDIA M2090 GPUs (1024 CUDA cores). The interconnection between nodes is based on an InfiniBand QDR full tree network.

[3] For 32 LPs, each one simulates 32 AAs; for 1024, LPs each one simulates one AA

8.3.1.3 Performances metrics

In this section, we propose to analyze the performance of the considered simulators according to two metrics: the simulation runtime and the synchronization delay.
Simulation Runtime The simulation runtime is a generic metric which gives an overview of the global efficiency. It consists of the measurement of the needed time to achieve the totality of a given simulation. Figure 8.4 presents the average simulation runtime according to each LPs configuration. In what concerns the $NS-3$ simulators, we observe that its scalability seems respectable. In particular, we note that its runtime is approximately divided by two each time we double the computing power. In contrast, the basic CMW configuration is up to $5x$ faster with 32 LPs but the difference decreases when the number of LP increases. This is mainly due to the limit of cunetsim when the simulated population per LP is relatively small. To focus on the contribution presented in this chapter, we compare the difference between the basic CMW and the GP-CMW. First, we observe that for a small number of LPs'(32), the GP-CMW is as efficient as the basic model, which demonstrate that the gain compensates the inherent overhead due to additional management processes. Second, we observe that the GP-CMW model scale better when using larger distribution until becoming up to $2x$ faster when using 1024 LPs. If we highlight that both experimentats handle the same number of event, use same models and give the same output, we can assert that GP-CMW management model is effectively the most adequate solution for large scale hybrid simulation.

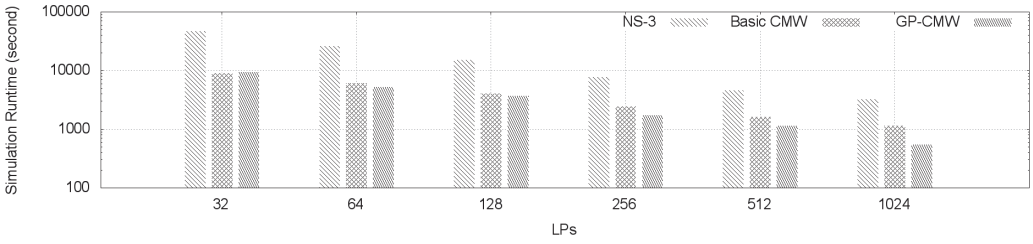


Figure 8.4: The simulation runtime of the studied simulators:

While both basic-CMW and GP-CMW models seem efficient under small scale conditions (32-64 LPs), the basic-CMW model presents major efficiency decrease when targeting a large number of LPs. In contrast, the GP-CMW scales well until 1024 LPs.

Synchronization Delay
We define the synchronization delay for each WU as the time difference between the first received acknowledgment and the emission of the release of the next WU. The value is computed at the coordinator level for cunetsim and as the average of all LPs for $ns-3$. To highlight the variation of the delay during the experiment, we rely on a statistical representation that summarizes the average value, the max, the min and the variance.

Figure 8.5 highlights the synchronization delay of the $ns-3$, the basic CMW model and the GP-CMW one, according to the increase of the number of LPs. We note that the synchronization delay of $ns-3$ increases continuously between 32 and 256 LPs. The delay becomes significant and unstable when the simulation includes 256 LPs. Thus, the average delay is about $3ms$ while the difference between the max and the min value is up to $6ms$. This major variation is due to the load divergence between LPs that increases, and reaches a peak at 256 LPs. However, when the simulation uses respectively 512 and 1024 LPs, the synchronization delay decreases until becoming $1ms$ in average. We remind that $ns-3$ uses a direct binary synchronization based on a lookahead mechanism. Accordingly, it is clear that this mechanism scales well and ensures a respectable synchronization delay on average. In contrast, we observe that the basic CMW is sensitive to the number of LPs: the average synchronization delay increases from $0.3ms$ to $3ms$ while the maximal delay during a large scale simulation reaches $4.5ms$. This behavior highlights the absence of any priority management mechanism and indicates the limitation of the basic CMW model in terms of scalability. The GP-CMW (that includesthe PAL) presents a stable behavior. The average delay oscillates between $0.3ms$ and $0.5ms$ while the maximal delay does not exceed $1ms$ despite the number of LPs. We also notice that the GP-CMW model is the most stable framework whatever the scalability conditions.

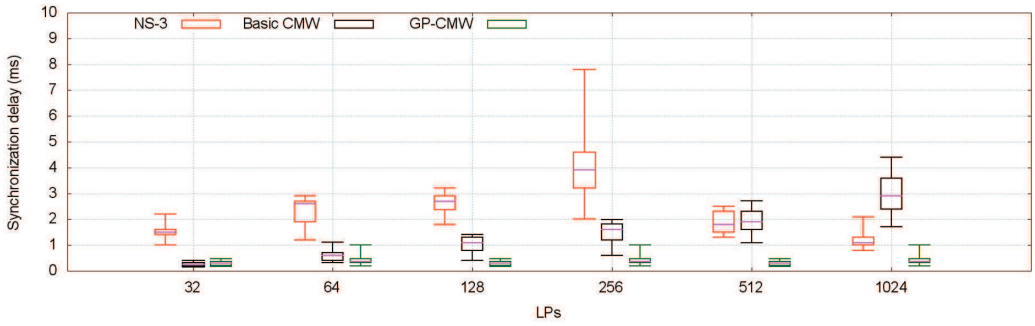


Figure 8.5: The synchronization delay as a function of the number of LPs

The synchronization delay of the $ns-3$ and the Basic-CMW models is sensitive to the the number of LPs. In particular, the synchronization delay of the basic-CMW model increases continually. In contrast, the GP-CMW maintain a stable average synchronization delay regardless of the number of LPs

8.3.2 Inherent Performance Evaluation

To study the performance of the GP-CMW model, we propose to compare its behavior with that of the original model. Previous works that study the CMW model [17, 16] validate a significant gain compared to existing software models. However,

the benchmarking scenario relies on a simplified grid network with flooding protocol. In this work, we propose to model a popular massively multi-player online game named *Command and conquer, Tiberius alliance*. The experiments are achieved using the Cunetsim framework, where we implement the game models as specified in section 8.3.2.1. The software and hardware platform are identical for both models which guarantee a fair comparison. The simulation scenario and setup are summarized in section 8.3.2.2. In order to highlight the efficiency of the GP-CMW model, the proposed comparative study focuses on four metrics: the simulation runtime, the synchronization delay, the communication latency and the hardware usage rate.

8.3.2.1 Game Model

The 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 it has one initial base with limited resources and must develop its base, army and defense. Players can create, integrate or leave an alliance. The alliance members share the controlled resources and infrastructure. However, the alliance size is limited to 50 players which means that all alliances are in direct concurrence to control resources and infrastructures and infinite the world. Moreover, alliances can define a diplomatic relationship with others. The development of players can rely on free available resources -limited in time- or buy resources packets using EA [39] funds. The Financial stock of the game is to sell such packets during the life of a world, which may last up to one year. Predicting the evolution of a world is painful, using simple analytic models since there is a large number of unpredictable parameters, which may influence the game process. For example, a war between two groups of alliances may double the number of connections, the connection duration as well as the number of sold packets. During the three monitoring months of the experimentation, servers were down for maintenance more than 20 times (approximately 30 minutes each). Furthermore, the main web site was down twice for more than 3 hours and several worlds were down without previous notification. Therefore, we can expect that it was difficult for EA to predict players' behavior even if they have all available traces. The interest of modeling this game is twofold: first, event generation of such simulation is dense enough and sustainably random which presents a good candidate for a very large scale and realistic simulation. Secondly, the rapidity of the cunetsim framework allows the publisher to predict the impact of marketing strategy modifications in real-time.

To model the game, we define two workers' classes: a player and an alliance. Concerning the player, we define three patterns which describe its behavior: mobility, communication and connection. Furthermore, the alliance behavior is defined by two patterns: the diplomatic pattern and resource control. To generate the corresponding patterns, we supervise the behavior of 1000 players and 30 alliances which

evolve in French worlds number 2,6,7 and 10.

8.3.2.2 Experimentation Scenario and Setup

The goal of the proposed experimentation is to predict the servers load and life on the one hand, and players' behaviors, on the other hand. Based on the Cunetsim framework, we define two referential scenarios: the first models one hundred worlds during one year with a granularity of one minute, which is equivalent to 525600 rounds and the number of players in each world is randomly chosen between 2 and 5 thousands. This scenario represents a medium simulation load. The second scenario is similar, except that the number of players in each world varies between 25 and 30 thousands, which represents a very intensive simulation load.
While each world is represented by one independent ELP, all simulations are synchronized at each round. This approach allows the user to predict the impact of any modification simultaneously in all the worlds. The simulation infrastructure is based on three workstations, each of which includes two NVIDIA GPUs: GTX 680 and one Hexa-cores CPU (i7 3930k). The operating system is the Ubuntu 12.04 LTS. The CUDA API version is the 5.0. We also use the PGI suite V13 as a compilation solution for heterogeneous targets. The hyper-threading feature of the CPU is activated. The PCIE generation is updated to the third standard.

8.3.2.3 Performance Metrics

Previous works related to the basic CMW model [17] discussed its efficiency compared to existent parallel and distributed simulator architectures that target multi-cores hardware. Therefore, we focus this study on a comparison between the basic and the GP-CMW models. We consider four referential metrics, according to two different measurement routines: to analyze the simulation runtime and the hardware usage rate, we use the average value of 10 experimentations. However, regarding the synchronization delay and communication latency, we have chose a representative execution with minimal ambiguity.
Simulation Runtime
The simulation runtime is a generic metric which gives an overview of the global efficiency. It consists in measuring the needed time to achieve the totality of a given simulation. In the proposed case, this metric is particularly fair since we use the same simulator, models and infrastructure. In this section, we consider the first and the second scenarios which differ by their scale and load. In the first scenario, the total number of simulated workers is about 350k player and 10k alliances. The simulation generates approximately 2 Tera-events. The second scenario is more complex, since it involves about 2 million workers and generates up to 50 Tera-events. Figure 8.6 emphasizes the simulation runtime of both models where we compare four configurations of the simulation platform: (1) the basic CMW model without any optimization. Its measurements present the referential. (2) The basic CMW model with the PAL. (3) The basic CMW model where we activate HAL. (4) Finally, we use the totality of the GP-CMW including the two optimization layers.

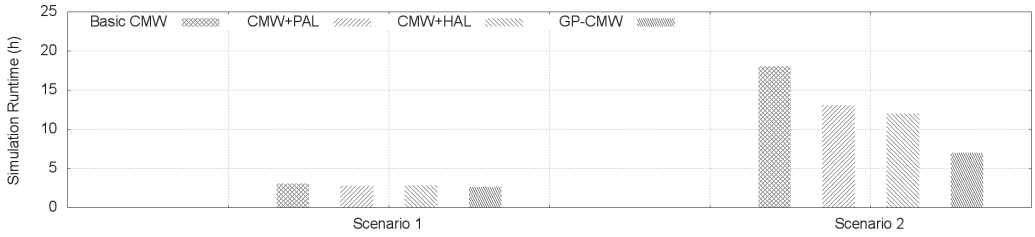


Figure 8.6: The impact of the PAL and HAL on the simualtion runtime:

We observe that the GP-CMW model is up to 20% more efficient than the basic one under medium load and up to 2.3x faster under extreme load. The impact of the HAL is more meaningful since it maximizes the usage of available hardware.

The referential runtime of the first scenario is about 3 hours. Relative gains of the second and the third configurations are respectively 12% and 7%. Therefore, we can admit that each optimization layer provides a measurable gain. Nevertheless, the HAL seems more critical. This aspect is comprehensive since the H-scheduler uses the totality of available resources while the PAL optimizes existing flows. In practice, the HAL uses the totality of the CPU cores for sporadic events that maximizes the hardware usage rate. Finally, the combination of all these improvements results in a gain of 19%. This gain proves that the proposed optimizations are compliant and may work simultaneously in perfect harmony. With regard to the second scenario that presents an important simulation load, results are more differentiated. The referential runtime is about 18 hours and relative gains of the second and the third configurations are respectively 50% and 100%. In this measurement series, the impact of each layer is visible: we notice that the most valuable optimization is definitively the HAL. The PAL participates on the stability of the simulation especially under utmost load. This observation is confirmed with the impressive gain of 260% of the GP-CMW model compared to the referential one. Its results prove that the introduction of the two optimization layers allows maximal stability and efficiency.

Synchronization Delay

In the proposed massively multi-player game, we define the synchronization delay as the required time to synchronize the beginning of any round in every simulated world. The measurement is achieved on the coordinator level according to the elapsed time between the first and the last received acknowledgment. Figure 8.7 presents the synchronization delay of the four configurations described previously in section 8.3.2.3. We rely on a statistical representation that summarizes the average value, the max, the min and the variance. With regard to the first scenario, we observe that the average delay is relatively constant for all configurations between 0.26 ms and 0.28 ms. However, we notice that the stability of each system -defined as the deference between the maximal and the minimal values - is improved when

the PAL is activated (Config 2 and 4). This conclusion is confirmed with the second scenario where we notice that the average synchronization delay without the PAL is about 0.48 ms and reaches 2 ms in several picks. In contrast, configurations that include the PAL maintain the same delay of the previous scenario (between 0.26 and 0.28 ms). Furthermore, the variance of these configurations remains less than 0.19 ms while the basic CMW model reaches a variance of 1 ms. According to these results, we conclude that the introduction of a priority management solution into a distributed simulation framework that relies on a heterogeneous computing resources is mandatory to maintain the stability of the synchronization mechanism and more generally the stability of the simulation system under variable conditions. It also provides a real gain in terms of simulation runtime.

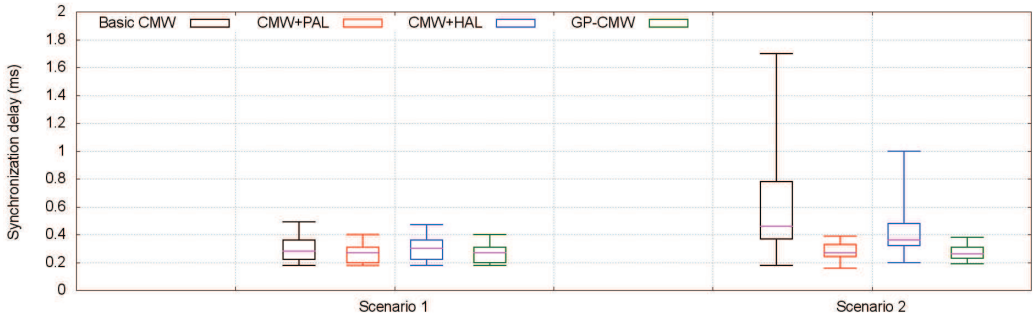


Figure 8.7: The synchronization delay as a function of the scenario and the configuration

To highlight the variation of the synchronization delay, we use the candle-sticks representation: 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.

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) distinct ELPs known as External communication and (ii) simulated entities of the same ELP known as internal communication. We compute two values for each worker and master (ELP). We rely on a statistical representation that summarizes the communication latency for both scenarios.
Figure 8.8 presents the measurement of external communication for the four configurations presented in section 8.3.2.3. We notice that the basic CMW requires in average about 5 ms to ensure the transmission of a message between two distinct ELPs. In contrast, configurations that include the PAL are up to 5x faster. We note that the HAL remains neutral since it does not induce any additional overhead. We conclude that the aggregation policy that we introduce to manage external commu-

nication is particularly efficient and allows to increase the inter-ELP communication feature.
Figure 8.9 shows the measurement of internal communication for the four configurations. We note that the basic CMW model requires in average about 5 us to ensure an internal message transmission while configuration that relies on the HAL achieves the same work up to 10x faster. We note that the PAL does not have any impact on the inside communication latency. In fact, this significant gain is mainly due to the usage of the direct access memory feature that allows any worker to access the input-buffer of other workers if authorized without an intermediate request to the CPU.
Accordingly, we conclude that the introduction of the HAL and the PAL provides a significant gain in both internal and external communication latency. We also assert that combining these features allows a maximal stability.

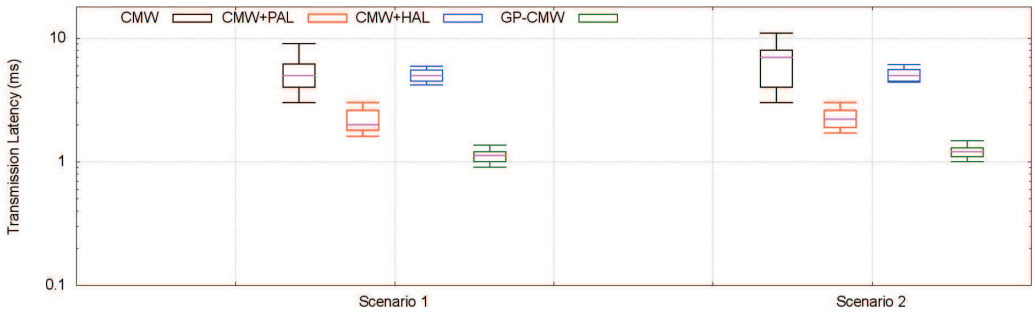


Figure 8.8: The average external communication latency:

We observe that the PAL significantly reduces the outside communication latency. The aggregation approach provides is up to 5x faster than direct communication

Hardware Usage Rate The hardware usage rate is a generic term that regroups the measurement of each component usage during the software execution. With respect to this study we focus on four components: the CPU, the GPU, the RAM and the GDRAM. The measurement is referred to the maximal load and presented as a percentage. We rely on the OS primitives to evaluate the usage, and we compute the average value during the simulation.
Figure 8.10 presents the average CPU usage rate during the simulation. We notice that the GP-CMW model is able to maximize the usage of all available cores (6 cores) while the basic CMW model uses one CPU core to manage the simulation. This is mainly due to the efficiency of the HAL that integrates the hybrid scheduler which may schedule each event on the most adequate target. In particular, the hybrid scheduler uses the CPU to execute isolated and sporadic events.
Figure 8.11 presents the average GPU usage rate during the simulation. It has to be mentioned that we consider the average value of the two GPUs of each machine. We notice that the GPU is correctly solicited by both models since its usage rate

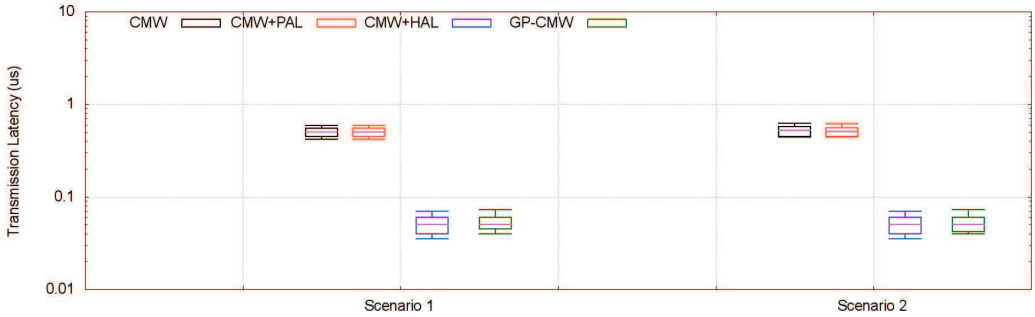


Figure 8.9: The average internal communication latency:

We observe that the HAL reduces significantly the outside communication latency. This is mainly due to the usage of the improved direct communication inside each addressing space.

exceeded the 70%. However, the GP-CMW model reaches up to 90%. The main reason for this non-negligible gain is the hybrid scheduler that avoids executing ungrouped events on the GPU.
Figure 8.12 presents the main memory load during the simulation. As expected, the GP-CMW model requires up to 50% more memory than the default one. This is mainly due to the usage of intermediate buffers, used to thin the events' flow. Finally, Figure 8.13 presents the GPU memory load (GDRAM). We notice that both models maximize the usage of available memory. In fact, both models target to maximize the efficiency of this memory (the CMW is basically designed to target simulation over GPUs).
Accordingly, the GP-CMW model provides a real gain in terms of smart usage of heterogeneous resources. In particular, the HLA (that integrates the hybrid scheduler) provides an efficient separation between the event modeling and execution which allows the framework to select the most adequate target dynamically. In contrast, the default model switches all the events to the GPU (or the CPU in the legacy mode) which limits the simulation flexibility. We conclude that the GP-CMW model, presented in this work, maximizes the efficiency and flexibility of the system.

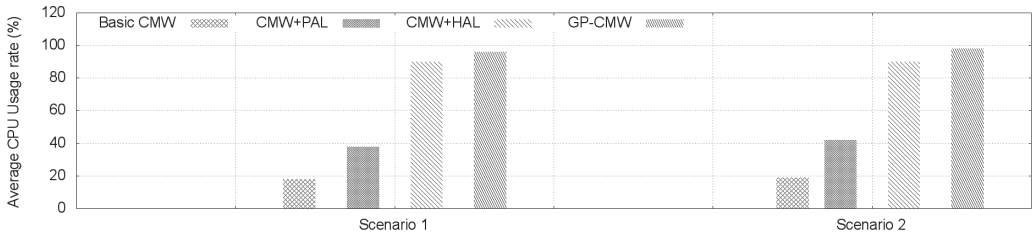


Figure 8.10: The average CPU usage rate:

the GP-CMW model ensures the maximal usage rate, according to the usage of the efficient H-scheduler whitch uses the CPU as a simulation resources

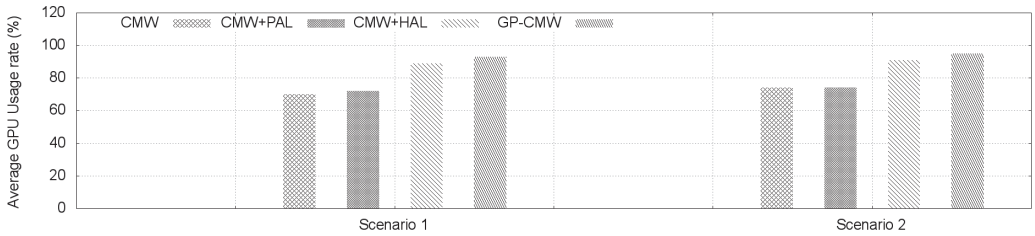


Figure 8.11: The average GPU usage rate:

All configurations provide a reasonable GPU usage rate since the basic CMW model is designed for GPU-based simulation. However, the event switching policy of the H-scheduler maximize the usage of the GPU also since it reduce the execution of non grouped events in such resource.

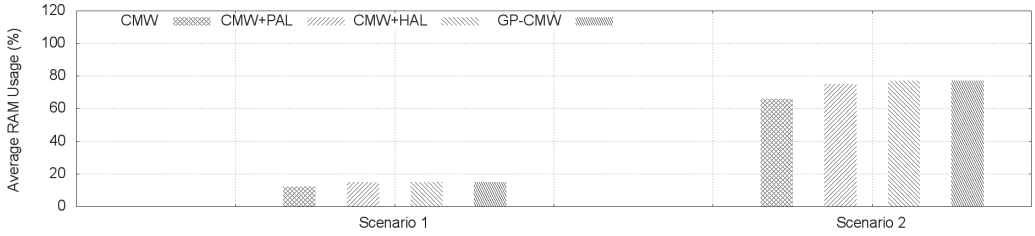


Figure 8.12: The average RAM usage rate:

As expected, the introduction of intermediate buffers increases the usage of the memory. The overhead is about 25% which may be significant in several cases.

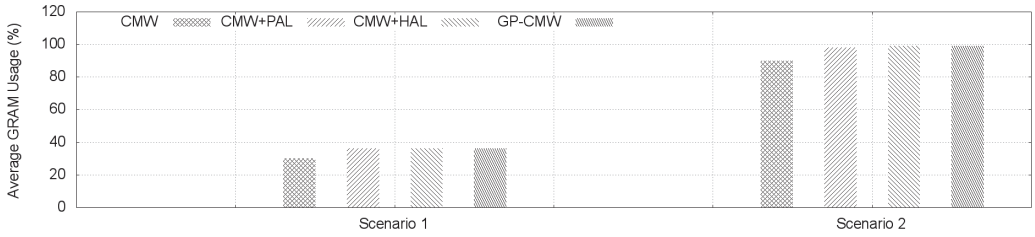


Figure 8.13: The average GRAM usage rate:

The used memory is relatively similar since there is no significant variation between their usage logic. However, since the GP-CMW maximize the GPU usage rate, it maximize also the GRAM usage.

8.4 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 that time, computer hardware had evolved considerably, and the computing power doubles each two years almost. Nevertheless, the complexity and the scalability requirements of digital simulation increase rapidly. Consequently, the sequential simulation appears inadequate with that requirements since it provides a limited tradeoff between the computing time and the simulation extent. Thus, there is a consensus on the issue: *To increase the simulation efficiency and scalability, community needs to avoid the weakness of centralized solution.* On the other side, the computer architecture -that has long remained invariant (CPU-RAM HDD)- has changed radically in the last decade: the computer becomes a group of heterogeneous resources that collaborate to perform a task. Ignoring that improvement results on a low hardware usage rate which increases the inherent cost of any simulation due to additional ressources an power that must be deployed to bridge the waste of resources. To deal with that issue the ingenuity of researchers has proposed a variety of optimizations that we can classified into four classes[110]: 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 [83]. The scalability remains an issue in the flat design when the number of LPs increase [47]. In the literature several optimizations, such as the lookahead [46] and the opportunistic and combined synchronization [110], 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 synchroniza-

tion and job assignment processes. The well-known example is the master/worker model compatible with meta-computing systems [95]. 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 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 [144]. 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 [4], 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 [106, 26], 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. 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 available cores to execute in parallel future events[85] . However this approach relays on a central events list and one scheduler, which remains the bottleneck when targeting larger CPUs (e.g. INTEL MIC with 80 cores) [123]. A dedicated GPU scheduling approach was proposed in [15], 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 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 worlds per clock cycle maximize the performance of the re-wrote routines). A smart usage of that capabilities allows a significant performance gain [8]. 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. For example, Steenkister et all [19] used the FPGA as

a signal accelerator for wireless network simulation. The OpenAirInterface [18] 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 the same context, the GPU immerges as a carrier solution that combines programmability and large computing power. Nevertheless it requires a specific software architecture since its programming model is not fully compliant with the x86 architecture. Despite this limitation, Perumalla et all [108] demonstrate the feasibility of using the GPU as a simulation context and several works proof its efficiency as signal processing accelerator [5, 10]. 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 [32]. In particular the XEON Phi co-processor [64] provides up to 64 CPU computing core per device. That solution seems promising and several recent works assert that its development-cost/gain tradeoff is interesting [124, 58].
In contrast with these fundamental approaches, there are a fourth category that propose to combine their advantages to reach a maximal scalability and efficiency levels. We denoted it as the hybrid category as a reference to its combinatory nature. It introduces a revised software architecture while using massively local optimization and optimized libraries. Moreover, the usage of heterogeneous computing resources is considered to maximize the performance. In this perspective, $ns-3$ is a well-known network simulator that combines new software architecture with massively optimized code [76]. Further, new frameworks that rely on virtualized resources combine both architectural and local optimization to perform optimal usage of virtual and real resources [149]. Nevertheless, GPU and hardware acceleration solutions in general are weakly considered in such approaches. The basic CMW model [17] contemplates the hybrid approach differently since it combines modified software architecture with the usage of heterogeneous computing resources including multi-cores CPUs and GPUs. However, it was natively designed for network simulation and presents some stability and extensibility weakness. Accordingly, the GP-CMW, that we detail in this chapter, is a generalization of the basic model that can be used for general purpose simulation and increases the stability and the extensibility of the system.

8.5 Conclusion

Parallel and distributed simulations are considered as the main approach to improve the speedup 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 in view of a large number of events. In this context, one of the central challenges is to find an optimal tradeoff between

communication and data locality. On the other side, meta-computing systems composed of several interconnected heterogeneous systems emerge as a real alternative to traditional expensive and complex data-center. However, the usage of such infrastructure requires maximizing the interaction within each computing node while minimizing the communication overhead among the network.
To cope with these specifications, we propose a three-level software architecture for general purpose simulation software. The proposed architecture extends the commonly used master-worker model by introducing a third top-level process denoted as the coordinator and two optimization layers respectively: the priority abstraction layer (PAL) that maximizes the priority of the control plan and the hardware abstraction layer (HAL) that ensures the execution of events over heterogeneous computing resources. The new model is denoted as the general purpose coordinator-master-worker model and relies on two fundamental concepts: 1) the separation between control and data flows. 2) The disassociation between event modeling, scheduling and execution.
In contrast with the basic CMW model, comparative evaluations prove that the GP-CMW model provides a significant stability, maximizes the hardware usage rate and increases the efficiency. In particular, it is able to handle a very large scale simulation up to 2.5x faster than the default model under the same conditions. Accordingly, we can assert that the separation between events' description and execution on the one hand and between control and simulation plan on the other hand, presents a promising perspective when targeting efficient simulation over meta-computing systems.

Chapter 9

Study Case of PADS Methodology Deployment: NS-3

9.1 Introduction

The main research topic of the thesis is the scalability issue of discrete event simulation (DES) and particularly the challenges of efficient parallel and distributed discrete events simulation. We choose to consider the particularity of recent hardware trends such as the GPGPU computing over heterogeneous resources. In the contribution part of this thesis, we detail a new events management model that separates the event from its descriptor. That feature allows the grouping of several cloned event into one descriptor which simplify the parallel scheduling process. In addition, we propose a hybrid events scheduling mechanism targeting the parallel simulation over heterogeneous resources. This mechanism allows new opportunities in terms of flexibility and extensibility of the simulation framework while increasing the usage rate of available resources. Furthermore, we propose a general purpose coordinator-master-worker model that addresses the problem of large scale parallel and distributed simulation (PADS) over meta-computing infrastructure. This model aims at maximizing the activity inside each computing node and minimizing the interaction of distinct entities in order to reduce the global overhead . These contributions are mainly validated using the Cunetsim simulation framework. Cunetsim is a lightweight network simulator that we developed from scratch at EURECOM to study the feasibility of our hypothesis. While this framework is sufficient as a proof of concept when targeting simplified benchmarking scenario, it is not widely used by the community in order to be recognized as definitive validation support. In fact, Cunetsim implements simplified models of a reduced number of patterns (mobility, connectivity, packets services ...) and it lacks realism compared to existing network simulator such as NS-3.

Even if the proposed contributions seem promising, it will be difficult to convince the community to use the cunetsim framework. In fact, recurrent answers when we suggest the usage of cunetsim to our colleagues include(1) the required effort in term of development. (2) The learning phase due to the modification of scenario description that generates parallel events and (3) the limited usage of very large scale simulation in their work makes obsolete the usage of a new framework, just because it is faster. Meanwhile, one of our goals is to release Cunetsim as an open source project to the simulation community. Thus, it is important that we prove the feasibility of their integration into existing frameworks that already have a large public and a

significant notoriety. Naturally, we choose the famous network simulator NS-3 as a study case to demonstrate the benefit of the techniques developed in Cunetsim.

9.2 Overview

The $NS-3$ simulator is the newest project that extended the popular $NS-2$ simulator that has long been widely used as a validation platform for research and education on networking systems. The main goal of the $NS-3$ project is to produce a discrete-event network simulator with an emphasis on layers 2-4 of the network stack. The project aims to provide fundamental solution for $NS-2$ limitation in particular, in terms of design weakness, scalability and extensibility. The $NS-3$ team identifies a number of factors contributing to the scalability limitation of its predecessor, including software architecture and events managements. However, the fundamental bottleneck is the execution on a single processor. In contrast, $NS-3$ is natively designed to support parallel and distributed simulation using a federated simulation developers kit and a ghost-node approach. It uses also a middleware layer (Runtime Infrastructure- RTI) that allows direct communication between simulated nodes. $NS-3$ relies on a conservative synchronization approach: no federate in the parallel simulation will ever process an event that would later have to be undone due to receiving messages in the simulated past [61]. A multi-core parallelism work is proposed on [126]; the concept uses a multi-threaded execution context, which is supplied by one LP. A comparative evaluation of parallel simulation using ns-3 highlight the importance of using the multi-threading to improve the simulation runtime [81].

However, all these optimization ideas consider the backward compatibility with two design rules: (1) the event scheduler does not have any knowledge about the event except its timestamp and (2) one event is relative to one simulated entity. In this context, it has to be mentioned that $NS-3$ is split between a core simulator part and a models part [73]. The simulator core looks like an independent discrete event driven simulator that considers the models part as a foreign layer which generates events. This layer-separation between the events generation and the simulator core increases the consistency of $NS-3$ architecture. Nevertheless, the event management model is still conventional and becomes the bottleneck under large scale conditions [143]. The parallel event scheduling over multi-core CPU had been proposed in [73, 74] and can be used on the $ns-3$. Though, the event scheduling remains a significant bottleneck when targeting large scale simulation even if we rely on a modified event scheduler that supplies several threads.

In the remainder of this chapter, we first highlight the importance of the event scheduling on $ns-3$ as a system bottleneck in section 3. In section 4, we propose different solutions that address the scheduling issue according to previous experience on the cunetsim framework. In section 5, we propose a comparative evaluation of that solutions using a partial implementation of each one. We conclude the chapter on in section 6.

9.3 Events scheduling on NS-3

$NS-3$ implements a conservative and sequential event scheduler that sorts the future events list according to the simulation timestamps in order to determine the scheduling order. Since a layer-separation design is used, the scheduler does not have any prior knowledge about event characteristics. Thus, when the event generator generates a group of identical events that may be executed in parallel (i.e. using a simple loop for), the scheduler will handle them individually one per one (Figure 9.1).
Distributed $NS-3$ is available since 2009 (the version 3.8). It implements a conservative parallel discrete event simulator. It considers special point-to-point links that connect network across logical processes (LPs). Each LP manages one execution context and communicates with other LPs using MPI interface. Figure 9.2 models two LPs, each of which manages one independent future events list. The main limitation of that concept remains in its incapacity to handle heterogeneous computing resources. Moreover, it requires the creation of several LPs to maximize the usage of multi-core CPU which implies in turn a significant overhead.

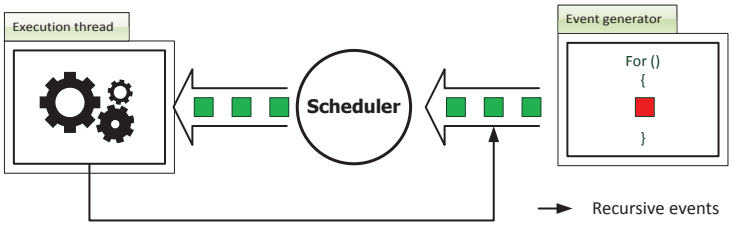


Figure 9.1: Default NS-3events scheduling approach:
The scheduler relies on a conservative method where events are sorted according to their timestamps and executed sequentially. The scheduler did not have any prior-knowledge about events nature except their simulation timestamps.

To highlight the events scheduling cost on the $ns-3$ framework under various loads, we propose to profile the software execution. For this purpose, we identify three software blocks: events generation, scheduling and execution. Events generation includes all software parts upstream the events scheduler while event execution is identified as the achievement of these events. We rely on the referential benchmarking scenario that we describe in chapter 6. The scenario models a simple network, where nodes are arranged in a grid topology. It includes one traffic source which generates 600 uniform packets every 1 second. Packet size is fixed to 128 Bytes. All nodes relay unseen packets after a delay of 1 second, thus flooding the totality of the network. The delay of 1 second models the propagation. Nodes do not provide any packet management services. Transmission and reliability are modeled on the channel using a fixed dropping probability which is identical on all links. The sender is the node with the lowest identity, and the receiver is the one with the highest

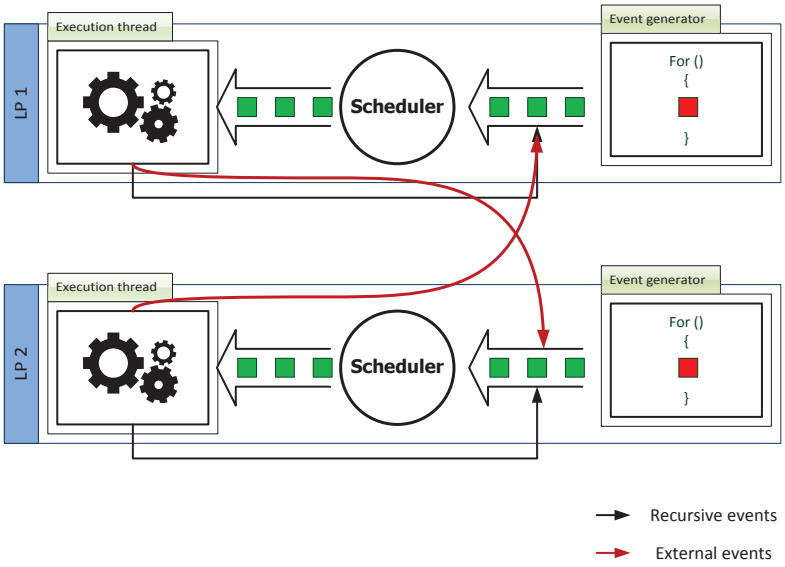


Figure 9.2: Distributed event scheduling approach in NS3:
There is a group of LPs, each of which achieves a part of the simulation independently. LPs exchange events using a remote infrastructure based on MPI. Received events are inserted into the events queue to be scheduled on the correct time. A lookahead mechanism avoid any causality rule violation.

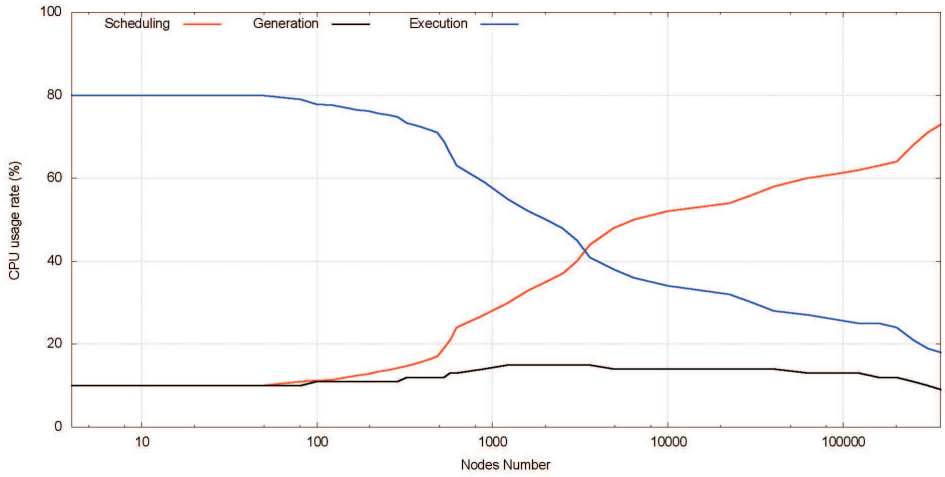
identity. To increase the simulation load we increase the number of simulated nodes since the number of events is proportional to that of nodes. Figure 9.3(a) present the normalized CPU usage rate of each software block as a function of the nodes number.

Figure 9.3(a) presents the simulation runtime of each software block during this experiment. We notice that the event modeling and scheduling impact remain low under small to medium load. In particular, when the simulation includes one thousand nodes, the event scheduling represents about25% of the CPU usage. However, the scheduling cost increases rapidly and reaches up to 50% for 10000 nodes and exceeds the 70% for 300k nodes. There are two aspects that impact the event scheduling: firstly, the event generation respects a given pattern where each node forwards a packet to almost three neighbors. Thus, the events number is linearly proportional to the number of nodes on the simulation. Secondly, recursive events present an important proportion of the total number of the handled events. That means that the event scheduler performs a large number of reordering/sorting operations to determine parallel events. These reasons explain why the event scheduling uses up to 70% of the total computing power. While these results show how expensive it is to handle a large number of events individually under such conditions, it has to be mentioned that the highest part of the processed events in this scenario are based on messages exchange. However, the $ns-3$ implementation that uses a smart pointer concept minimizes the real exchange of identical message. Indeed, the processing of that event becomes extremely fast compared with their scheduling cost. We conclude that the measurement presented in this figure may schematize a maximal asymptote for general purpose simulation.

Accordingly, the scheduling is definitively identified as a critical bottleneck that we must address to improve the simulation efficiency. In addition, specific modifications and optimizations are required to handle both resources usage maximization and scheduling complexity. Thus, we propose several event scheduling extensions that survey the state of the art in DES.

9.4 NS-3 Events scheduler extensions

To address the events scheduling issue, we propose to evaluate the efficiency of four distinct extensions, each one presents numerous advantages and drawbacks. The first extension is the explicit parallel scheduling through multi-threading. The idea is to allow the event scheduler to execute events that do not present a relative dependency in parallel using several threads. The second extension is the implicit parallel scheduling through OpenMP API. The idea is to modify the events generator such that the grouped events will be presented as a unique entry that includes into its source code parallel arguments and OpenMP Pragma. The third extension is the GPU parallel offloading. The idea is to modify the events generator such that the grouped events will be presented as a unique entry that includes into its source code parallel arguments and OpenACC Pragma. Finally, the fourth extension is based



(a) Evolution of the events scheduling cost in the first scenario: the scheduling cost increases as a function of the number of nodes. It presents up to 50% of the CPU usage when simulating 8000 nodes and reaches 70% for the maximal simulation load. Note that almost of the processed events of this scenario are relatively lightweight compared to their own scheduling cost.

Figure 9.3: Scheduling Cost as a function of nodes number

on the usage of a co-scheduler. The co-scheduler will catch all the parallel events and determines to use the most adequate execution target. These extensions were proposed in different previous works, but to the best of our knowledge this is the first work that provides an implementation of all of them using the same productivity framework; $ns-3$. Thus, we can expect reliable and comparable results.

9.4.1 The Explicit CPU Parallelism

The explicit CPU parallelism consists of the execution of *parallelizable* events on different CPU cores at the same time. The decision of which is parallelizable depends on the scheduling policy. The conservative policy suggests that we execute events in parallel if they have exactly the same timestamps. Thus, the unique dependency control is done according to the simulation time. This approach has the advantage of proposing an interesting trade-off between the decision cost and the runtime gain. In addition, it requires a minimal modification at the scheduling level without any modification from the user point of view.
Regarding the $ns-3$ scheduler, previous effort proposes to parallelize events execution according to a conservative policy [126]. However, there is no official implementation that supports this feature. Thus, we propose the following scheduling mechanism: first, an execution ***threads pool*** is generated and wait to be supplied. The scheduler fetches next event according to its timestamp and executes it on the available thread. Secondly, the scheduler continuously fetches and supplies events to the execution threads until reaching an event that presents a higher timestamp. In this case, the scheduler waits until the achievement of all previous launched

events, before restarting a new round. Figure 9.4 schematizes the explicit parallel scheduling where the scheduler supplies three execution threads.

Weakness and limitations:

The explicit CPU parallelism presented in this section has two main weakness: first it may induce a significant waste of resource due to the strict conservative approach that forbids the parallel execution of non-co-timed events. Secondly, the system relies on a central events scheduling path(FEL+ scheduler) and several execution threads. Accordingly, that central path becomes the system bottleneck when increasing the number of execution threads. In fact, this approach may reduce the overall simulation runtime but does not address the underlying issue of the event scheduling.

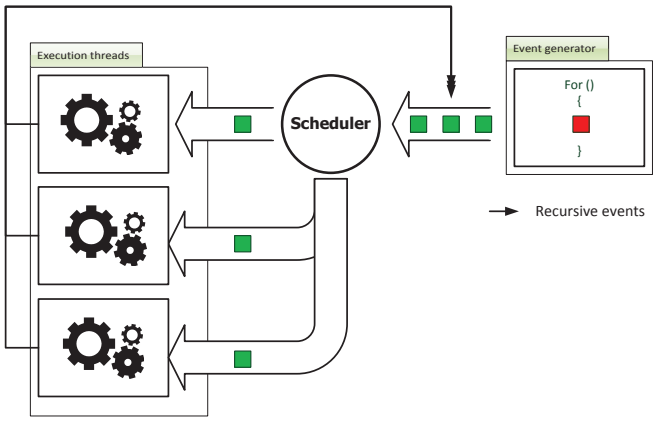


Figure 9.4: Explicit CPU parallelism:
The event scheduler detect all events presenting the same timestamps and schedule them to be processed as soon as possible. Recursive events are re-injected on the main event queue.

9.4.2 The Implicit CPU Parallelism

The implicit CPU parallelism aims at offloading the intelligence from the scheduler to peripherals components. Thus, rather than generating a large number of identical events and detecting that it is possible to schedule them in parallel subsequently, it is possible to generate one grouped event that will be scheduled as one and will be ungrouped at the execution. This approach has the advantage of reducing the number of scheduled entries while keeping the same number of events at the execution. Thus, the scheduling cost decreases proportionally to the number of grouped events. This approach is denoted as *implicit* in the sense that it is transparent from the scheduler point of view.

Regarding the ns-3 implementation, we propose to use the OpenMP framework as following: first, we modify the scenario structure such that the generation loop be-

comes integrated on the event. Secondly, we modify the event data such that all required data becomes an input/output parameters of the new event. Thirdly, we include the OpenMP Pragma in the event body. The Pragma is a pre-compilation directive that allows the OpenMP library to transform that part of the code into multi-threaded execution. Figure 9.5 depicts the implicit parallelism where the grouped event flow as a unique entry from the FEL to the scheduler. At the execution level, the corresponding number of threads is created to ensure the execution in parallel.

Weakness and limitations:

The implicit parallelism provides a significant gain and reduces the events scheduling cost, even under high load. However, its implementation requires a considerable modification on the simulation design.

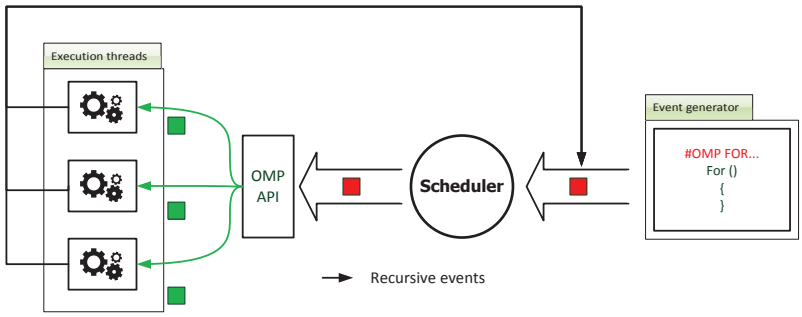


Figure 9.5: Implicit CPU parallelism:

The main modification happens at the generation and the event code where we include the corresponding loop into the event code. The principal gain remains on the scheduling cost that decreases proportional to the increase of the grouping size.

9.4.3 The GPU Offloading

The GPU offloading consists of the usage of the massive parallel computing power of the GPU to offload cloned events that may be processed in parallel without ambiguities. Similarly, to the implicit CPU parallelism, the implicit GPU offloading is transparent from the scheduler point of view. The main modification remains on the event code and the generation scenario. The advantage of this approach appears with large scale parallelism where we execute tens of thousands of cloned events.

Regarding the ns-3 implementation, we propose to use the OpenACC framework as follows: first, we modify the scenario structure such as the generation loop becomes integrated on the event. Secondly, we modify the event data such that all required data becomes an input/output parameters of the new event. Thirdly, we define the data management Pragma that define how *the memory transfer* will be achieved between CPU and GPU contexts. Afterwords, we include the *OpenACC* Pragma in the event code. Figure 9.6 schematizes the GPU offloading, where the grouped

event flows as a unique entry from the generation to the scheduler. It has to be mentioned that the GPU-scheduler of this diagram represents a software/hardware engines that works transparently, from the main scheduler point of view. It manages the creation and the termination of GPU threads.

Weakness and limitations:

The GPU offloading approach provides a significant gain in terms of runtime when combined with a good design of the event code. However, its main weakness is its sensibility to the tuning parameters. Moreover, the data transfer between both simulation contexts induces a significant overhead.

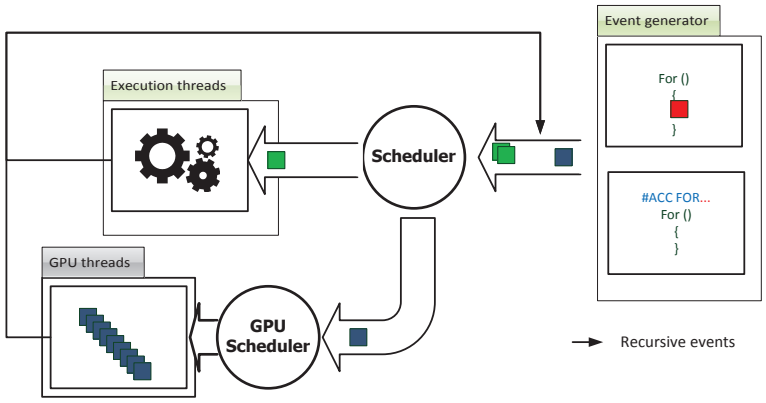


Figure 9.6: GPU offloading:
Similarly to the implicit CPU parallelism, the main modification happens at the generation and the event code where we include the corresponding loop into the event code. However, it has to be considered that the execution context is not the same which requires additional data management.

9.4.4 The Co-scheduler Approach

In previous sections, we had presented three optimization approaches that aim at maximizing the usage of available computing resources to increase the scalability and the efficiency of the simulation. In addition, two of them propose to reduce significantly the events scheduling cost based on a grouping policy. Each of these approaches presents several advantages and weaknesses that limit the overall expected gain. To synthesize these approaches into one efficient and all-in-one solution, we propose to combine them on the same simulation framework. Therefore, to maximize the scheduling efficiency we introduce a co-scheduler that manages both GPU offloading and implicit CPU grouped events (Figure 9.7). In contrast, the main event scheduler ensures the explicit CPU parallelism as detailed previously.
Regarding the ns-3 implementation, we propose to customize the events scheduler as follows: first, an execution threads pool is generated and waits to be supplied. One of these threads serves as the co-scheduler. Similarly to the explicit parallelism

mechanism, the main scheduler fetches events having the same timestamps in order to execute them in parallel. Nevertheless, the scheduler needs to identify which entry represents a grouped event and which represents a single event. Accordingly we introduce additional modification that breaks up the *layers separation* rule: rather than limiting the scheduler knowledge to the event timestamps, we also provide the knowledge about the event nature. On the implementation, we introduce new binary argument for all events, to indicate if this event is grouped or not. Thus, if the argument value is set, the entry is grouped and must be switched to co-scheduling thread; otherwise, it will be switched to one of the available threads.

Weakness and limitations:

Without doubt, the concept of combining all possible parallelism approaches in one framework is promising. However, it requires modifications in the simulation kernel, models, scenario and logic. In addition, proposed co-scheduler induces extra processing requirements for management operation. This overhead must be considered when evaluating the final gain.

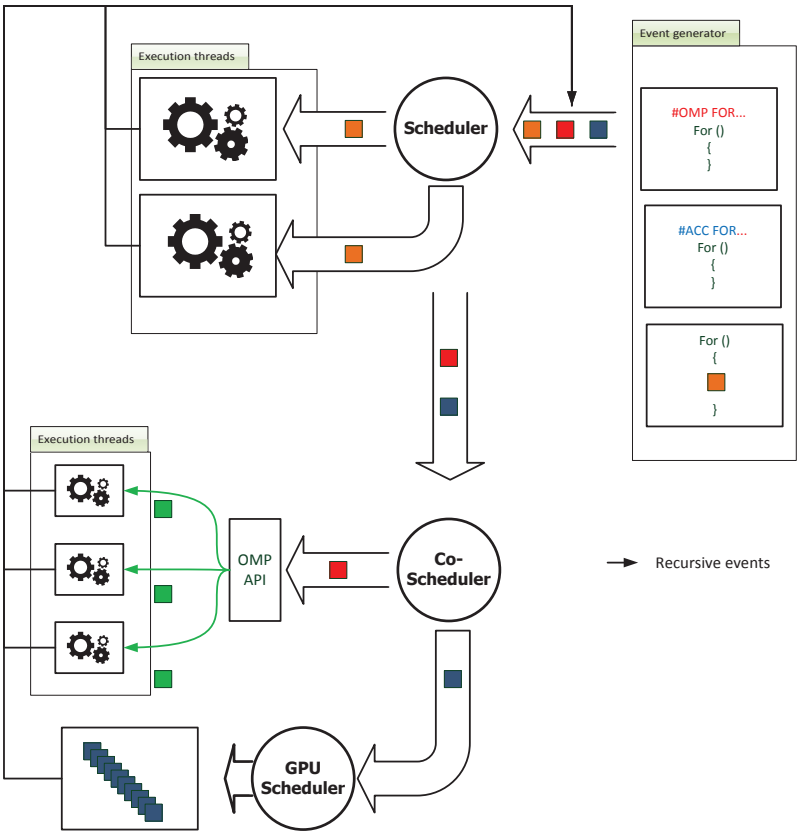


Figure 9.7: CO-scheduling approach:
The modification touch both the scheduling and the event generation, additional knowledge is required to manage the switching operation.

9.5 Comparative evaluation

In order to study the efficiency of each approaches, we propose a benchmarking simulation scenario using the $NS-3$ framework where we modify either the event scheduler or models or both as described in section 9.3 that simulates a group of identical node evolving into a 3D space (1600 * 1600 * 200 m). The simulation duration is 1001 rounds. Each node moves before each round and recalculates its connectivity set. There are six types of events: (1) Creating a message, (2) Sending a message, (3) Receiving a message, (4) Routing a message, (5) Moving and (6) Computing connectivity set. Concerning the first four events types, we used always the native ns-3 implementation. However, for the moving and connectivity estimation we provide three implementations: native one (for one node), an OpenMP implementation where we modify data management and an OpenACC implementation where we manage data transfer into the event body. To highlight the strength and the weakness of each approache we vary two parameters: the number of nodes from 1k to 500k and the number of execution cores from 1 to 48. Experimentation series are achieved on the TGCC Curie super computer [140]. We use the same simulation scenario presented in section 9.3; except that nodes move into a 3D space. In the remainder of this section, we highlight the runtime of five configurations:

1. *Distributed:* Is the official distributed version of $NS-3$, that uses one CPU cores per LP.
2. *Explicit CPU:* The configuration of the explicit CPU uses one LP. The number of execution threads is relative to the number of cores.
3. *Implicit CPU:* The configuration of the implicit CPU uses one LP. The number of execution threads is relative to the number of core(using OpenMP).
4. *GPU Offloading:* The GPU offloading configuration uses one CPU core and one GPU. There is also one LP. Accordingly its runtime does not change as a function of the number of cores.
5. *Hybrid Scheduling:* The configuration of the hybrid schedule uses one GPU and splits available cores between implicit and explicit execution.

9.5.1 Medium Load

We consider the simulation of 1000 nodes during 1001 seconds as a medium load of one execution LP. Figure 9.8 represents the results of five experimentation series, respectively using 1, 4, 8, 16 and 32 CPU cores. Results relative to 1 core shows the inherent gain or overhead for each configuration. We highlight that the distributed version and the explicit CPU parallelism are reduced to a simple $NS-3$ instance when the simulation framework is based on one CPU core.
When the simulation framework uses one CPU core, we notice that the implicit CPU parallelism is up to $2x$ faster than the native distributed and the explicit CPU

parallelism. The unique difference between the fist and the third configuration is the usage of the event grouping approach. Thus, we can assert that the expected gain of the event grouping is valuable and simplifies the scheduling work, even with a single processing core. In the same context, The usage of the GPU offloaded approach provides a gain of $10x$, even if only two types of events are offloaded on the GPU (moving and connectivity). This result consolidates the idea that the GPU is adequate for bottleneck offloading. Finally, the co-scheduling approach is almost as efficient as that using the GPU except that we can observe the overhead due to the additional computing tasks.
Regarding the scalability of each approach when dealing with multiple resources, the behavior differs considerably. First, the explicit CPU approach provides a significant gain with 4 and 8 cores. However, from the threshold of 16 cores, the gain becomes insignificant. Accordingly, it seems that the scheduler becomes almost the bottleneck when the number of execution threads becomes significant. Secondly, the distributed approach scales well with the increase in the number CPU cores. Thus, the simulation runtime decreases proportionally to the increase of the number of cores. It is necessary to note that the distributed version becomes up to $1.5x$ faster than the explicit-CPU parallelism when the framework uses 32 cores. This observation confirms that the event scheduler is a critical bottleneck that limits the scalability of the parallel simulation. Thirdly, the implicit CPU parallelism approach is clearly more scalable than the first and the second ones. In fact, its simulation runtime decreases proportionally to the number of cores until 16 cores. However, there is no difference between using 16 and 32 cores, which seems to be the scalability limit. Afterwards, the co-scheduling approach presents a significant gain regardless of the number of CPU cores and the used resources. Thus, it scales as good as the GPU offloading version when using one CPU core and outperforms all previous approaches when the number of cores increases. Moreover, it presents a significant scalability features and copes with a large number of CPU cores. This assumption is validated for up to 32 cores.

9.5.2 High Load

Figure 9.9 presents the results of a high load scenario that considers the simulation of 500k nodes during 1001 seconds. We realize five series of experimentation, respectively using 1, 4, 8, 16 and 32 CPU cores. In this figure, we observe that the explicit CPU approach is up to $2x$ faster than the distributed one when using 4 cores. However, this gain decreases continuously until it disappears when the simulation uses 16 cores. In such conditions, the distributed approach becomes more efficient above. Since both approaches rely on the traditional event description, we can definitively assert that event scheduler bottleneck cannot be avoided without a particular design on the event description.
Regarding the approaches that rely on grouped events representation(3,4 and 5), we observe that it is one order of magnitude more efficient than the ungrouped ones. Thus, using 4 CPU cores, the implicit CPU parallelism is up to $2x$ faster than the

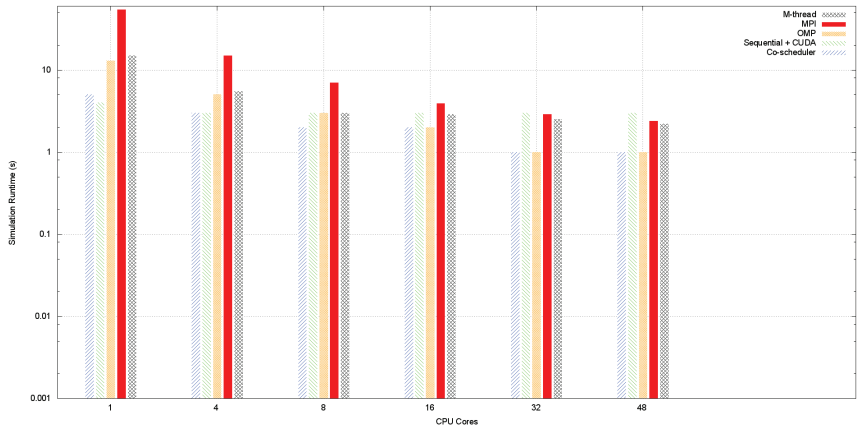


Figure 9.8: Simualtion runtime of each configuration as a function of the number of CPU cores (1000 nodes)

explicit one (multi-threaded), the GPU offloading is up to $5.5x$ faster, and the co-scheduling approach is up to $10x$ faster using the same resources. The limit of using a central scheduler appears when using 16 executions cores. Accordingly, combining the usage of ungrouped events with centralized scheduling is definitely identified as an inadequate approach to maximize the usage of heterogeneous computing resource while limiting the overhead. Afterwards, the co-scheduling approach that relies on a hybrid event scheduling method targeting in the same time the implicit/explicit CPU parallelism and the GPU offloading appears more efficient than all other approaches. It is up to $10x$ faster than the distributed solution and $15x$ than the multi-threaded one. Nevertheless,
To conclude these measurement series demonstrates that, parallelizing the execution could improve the simulation runtime but the scheduling remains a performance bottleneck. The distributed simulation minimizes that impact but introduce an additional overhead due to distributed management mechanisms. Grouping event approach is adequate for a particular event that can be grouped into one entry. This approach reduces the scheduling cost and simulation runtime. However, its scalability remains limited as the central path (scheduler+FEL) becomes the bottleneck when targeting large scale simulation. The same conclusion is relative to GPU based offloading except that it may provide more important gain when targeting massively parallelizable events.

9.6 Conclusion

In this chapter, we analyze the scheduling issues and limitations of the famous network simulator the $ns-3$. We formally identify the events-scheduling as the major bottleneck under large scale conditions. In particular, we highlight the scheduling cost of a very large number of events and the impact of events reordering. To deal with such limitations, we propose five approaches that we compare to identify the

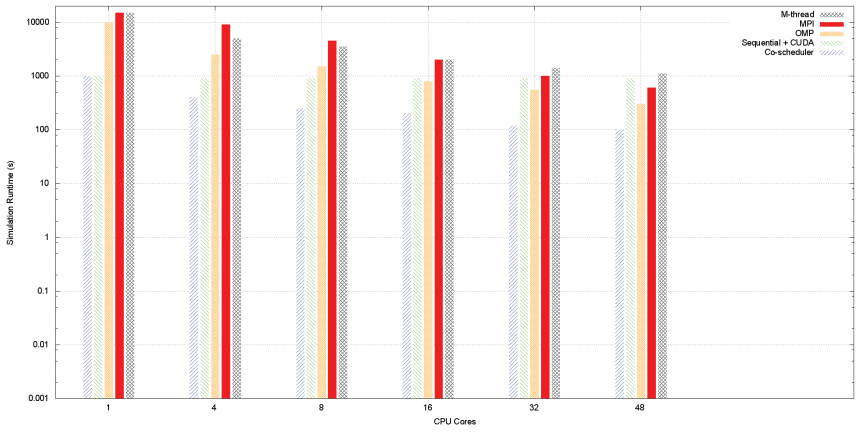


Figure 9.9: Simualtion runtime of each configuration as a function of the number of CPU cores (500K nodes)

most adequate with heterogeneous computing context. These approaches are the explicit CPU parallelism, the distributed approaches, the implicit CPU parallelism, the GPU offloading and the hybrid approach.
Comparative evaluation reveals the strength and the weakness of each approach. Above all, we observe that the multithreading is adequate to handle a reduced number of core but does not address the scheduling cost issue. In contrast, implicit CPU parallelism seems more flexible in term of resources handling but require an event modification which may be refused by the community. The GPU offloading approach provides a significant gain when applied correctly but requires also an event design modification and a dedicated data management model to ensure the data transfer between the CPU and the GPU. While such approach provides a significant gain in term of runtime, it violates the design pattern rules that separate between the simulation core and events generators. Thus, an architectural decision must be associated with the GPU usage.
Finally, the hybrid scheduling approach provides the maximal gain and combines the advantages of all the previous approaches. However, its concept relies on a cross layer design rather than a layered architecture. In particular, the intelligence of the system is proportionally distributed between events generation, scheduling and execution. There is no unwise software block in such model, all components must consider what will happen during the simulation and collaborate to ensure the simulation. This may shock conservative designer but seems inevitable to cope with the evolution.

Part III

Conclusion

Chapter 10

Conclusion

The scalability issue is a primary topic in the discrete event simulation that solicits several research works. In this thesis, we consider the problem of efficient large-scale simulation through heterogeneous hardware. In particular, we highlight the event management as critical limitation of DES. The most expensive operation in the management of the event is the scheduling of their execution in time. One characteristic of existing schedulers is the layer separation rule. It consists of the full separation between the scheduling, the generation and the execution of events. Thus, we propose to relax the layers separation rule in order to provide additional information to the scheduler and to reconsider the event management model.
During the management phases, an event is substituted by an event descriptor. Thus, from generation to the execution, the system handles events descriptors. The proposed approach extends the concept of event descriptor to group several events on the same descriptor. As a consequence, the management and scheduling cost are reduced, which in turn improves the simulation efficiency. Moreover, the proposed concept allows an efficient usage of the GPU as a main simulation context. In addition, we adopt a new event scheduling methodology that target to maximize the usage of available computing resources, including the GPU and CPU. The event scheduler dynamically switches each event/entry to an adequate execution target, based on the event type, load of the system and the specific characteristics of each execution target.
The proposed concepts are particularly adapted to share memory context. Thus, it requires specific software design to enable distributed simulation. Accordingly, we extend the well-known master-worker model to combine the advantage of parallel and distributed architecture on a three level model denoted as the general purpose coordinator master worker model. The idea is that the coupled master-worker resides in the same shared memory space and consists of one extended logical process. The coordinator is the top-level process and ensures master management operation such as the global synchronization.
The Cunetsim simulation framework is the main validation tool used in this thesis. The achievable scalability is assessed on the European supercomputer TGCC-Curie with 1024 LPs. Comparative results show that we can speedup the simulation up to 100x compared to CPU-based framework. We further validate the applicability of the proposed approaches, namely event-grouping and hybrid-scheduling, to the popular NS3 network simulator. According to the evaluation results, the optimized NS3 version is up to 25x faster than the basic one.
We assert that the event grouping approach combined with an optimized event mod-

eling, hybrid event scheduling, and coordinator-master-worker present a promising solution to achieve significant efficiency and scalability gain when targeting next generation of simulation framework.

Appendix A

Experimentation Methodology

A.1 Introduction

When we started delimiting the activities axes od this thesis, the first question that faces us is: are we doing reliable scientific experimentation? And from this interrogation stems another question: what is a reliable scientific experimentation in this research field? In fact, science computer and telecommunication are relatively new research activity compared to fundamental science such as mathematics and physics. Those historical sciences had developed during centuries a formal representation of scientific experimentation and validation methodology. On the other side, science computer community competes to be recognized as a science [37]. A major issue that confuses is the absence of standardization of the scientific experimentation as a validation methodology. On the other hand, it is important to bear in mind the phenomenal evolution of these sciences. In fact, during the last decades telecommunication and science computer areas converge continually. Emerging networks and technologies are the concrete representation of their fusion, where a simple smartphone, is a computer, a phone, a camera and embedded two or more networking interfaces. The complexity of such networks gives the experimentation a dominant position on their validation process.
Nevertheless, defining an experimentation standard remains a difficult task since any experimentation considering such Emerging networks evolves several uncontrolled parameters on one hand and includes several complex subsystems on the other hand. During the short life of computer science as a science, four experimentations methodologies dominate the literature: (1) there is the real world experimentation which relies on real *systems* that scientist study and analyze. However, such experimentation is difficult to reproduce due to the large range of uncontrollable parameters. (2)There is also the emulation which combines real components with modeled ones. It increases the reproducibility of the experimentation; however, just as the real experimentation it remains expensive and non-extensible. (3) There is the digital simulation which relies on the modeling of the simulated system with different granularity level. Its main limitation remains the realism level. In fact, the reliability of the simulation is a consequence of the reliability of the used models. (4) Finally, we distinguish the numerical models as that which use high level models, to produce an approximate view of what can happen. That methodology remains limited to primary usage and cannot be considered as a validation tool.
In what concerns the intern research activities of the mobile communication department, we mainly rely on the OpenAirInterface platform as a development and

validation tool. That platform provides the complete implementation of the LTE protocols stack and several peripheral instruments that ensure its casual usability. While existing ecosystem satisfy the requirement of rapid validation under development conditions, it seems insufficient to provide trustable scientific experimentation. Thus, we survey the literature to characterize scientific experimentation. In that sense, we identify four mandatory requirements: the reproducibility, the extensibility, the applicability and the revisability. Based in that analysis we propose a five-stepped methodology that aims to guarantee at least two requirements: the reproducibility and the revisability.

A.2 Scientific Experimentation

To define the boundaries of the experimentation methodology that we aim to create, we lead our fundamental work by searching scientific experimentation basis. In this sense, we summarize properties of scientific experimentation that we had retained as required. In computer science research filed, all development phases (Analysis, conception, test and validation, optimization) share one fundamental tool: the experiment. Nevertheless, the scientific feature of a computer science needs to be proved. In the literature reviews [54], we identify three properties that must be satisfied in order to be qualified as scientific:

- **Reproducible:** En experimentation is considered as reproducible if, for the same input, it produces the same output and presents the same behavior. Nevertheless, the notion of same may be ambiguous, in particular due to the usage of random numbers on the experimentation.

- **Extensible:** The extensibility of a given experiment may concern different aspects. In computer science, the number of involved elements, the density of the traffic/ activity. The extensibility is assimilated to the scalability on the area of network experimentation.

- **Usable:** The usability of a given experiment concerns the ability to operate on the results of the experiment and to identify the problem and/or the source of the problematic behavior. This property is particularly critical for large and complex system.

A.3 OpenAirInterface Experimentation methodology

In this section, we introduce our definition of the most adequate experimentation methodology which copes with OAI specifications and users requirements. First, we survey the proposed formal workflow that defines five sequential steps. Second, we present a general outlook of the corresponding implementation on both sides: user and developer.

A.3.1 OpenAirInterface Formal Experimentation Methodology

In order to guaranty the reproducibility of OAI experimentation, we propose a sequential workflow, where the output of each step is the input of the next[18]. Five consecutive steps are defined: *scenario description, configuration, execution, monitoring, analysis*, where each step is split into several sub-steps as explained in the following (see Figure A.1).

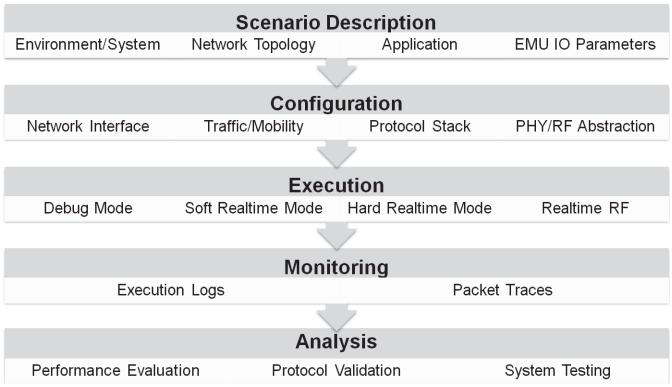


Figure A.1: Experimentation Workflow

We propose a five-stepped workflow; each step provides several mandatory/ or optional sub-steps.

Scenario Description This step builds a complete xml layout of an experiment. This step is splitted into four sub-steps: (i) *system/envirnment*, where system (e.g. bandwidth, frequency, antenna) and environement (e.g. pathloss and channel models) parameters are defined; (ii) *network topology*, where network area, network topology (i.e. cellular, mesh), nodes' type, initial distribution, and mobility model (e.g. static, random way point, grid) are set; (iii) *application*, where real application and/or emulated traffic pattern in terms of packet size and inter-departure time are defined; (iv) *EMU IO Parameters*, where supervised parameters (e.g. emulation time, seed, performance metrics) and analysis method (e.g. protocol PDUs and operation) are set.

Configuration This step defines a sequence of components' initialization based on the scenario description. It includes four sub-steps: (i) *network interface*, where the OAI IP interface is configured, (ii) *traffic and mobility*, where traffic pattern and mobility model parameters are set, (iii) *protocol stack*, where protocols are configured given the network topology and *PHY abstraction*, where a channel model prediciting the modem performance is configure.

Execution This step defines the execution environment for the simulation in order to synchronize nodes and run the experimentation. It includes three execution modes: (i) *debug mode*, (ii) *CPU mode* and(iii) *CPU-GPU mode*.

Monitoring This step defines how the experiment is monitored (passive and/or

active). It includes: (i)*execution logs*, where experiment traces and logs are collected, labeled and archived, (ii) *packet traces*, where protocol signaling is captured and stored during the experiment.
Analysis This step processes raw data and produces results and statistics. It includes three -non exclusive- analysis objectives: (i)*performances evaluation*, where the key performance indicators are measured and evaluated, (ii) *protocol validation*, where the protocol control- and user-plane signaling are validated versus protocol specification, and (iii)*system testing*, where the system as a whole is analyzed and tested.

A.3.2 Methodology Implementation

OAI is designed to be an independent experimentation platform that can be easily deployed in order to be shared by a given community. The minimal deployment model includes two independent categories: users and developers.
In what concerns the users' side, the workflow includes three servers: front-end, simulation and data-base. The front-end manages the interaction with the user based on a web interface. Authorized users can define an experimentation using an XML file or via an experimentation wizard that helps the user to generate its descriptive XML file. The XML includes mandatory parts the define respectively the scenario description, configuration, execution and monitoring parameters. When the front-end server receives the description file, it parses the XML and deduces the required resources and based on the user privileges the server proposes one or more execution schedule. If the user validates one of them, the experimentation is pushed on the execution queue of the simulation server. At the scheduled time, the simulation server takes action: it pops the description file and configures the simulation environment. Hence, it launches the execution and during the simulation runtime the monitoring process logs the output on the data-base server on real time according to the user scenario description. At the end of the simulation, the user receives an acknowledgment and may access and analyzes it output through the front-end server that embedded several analytic routines and algorithms.
In what concerns the developers' side, the procedure is quite different: its deployment relies on two servers: front-end and SVN. A developer can at any time get the last valid version and compiles it for internal usage. However, commit a contribution requires four validation steps: first, it must validate the quality of the proposed documentation through an automatic parser. Second, it must validate its unitary correctness through a list of compilation protocol that validates the binary across all supported software and hardware platforms. Third it realizes a primary integration test that validate the global correctness. That test relies on a group of referential scenarios that had a known output. Finally, the contribution is submitted to the server in order to pass the final validation. The development process is composed of several consecutive and short development rounds, each of which finish by an integration phase. The final integration phase validates the correctness of all new contributions that had been submitted during the last round. If there is an incom-

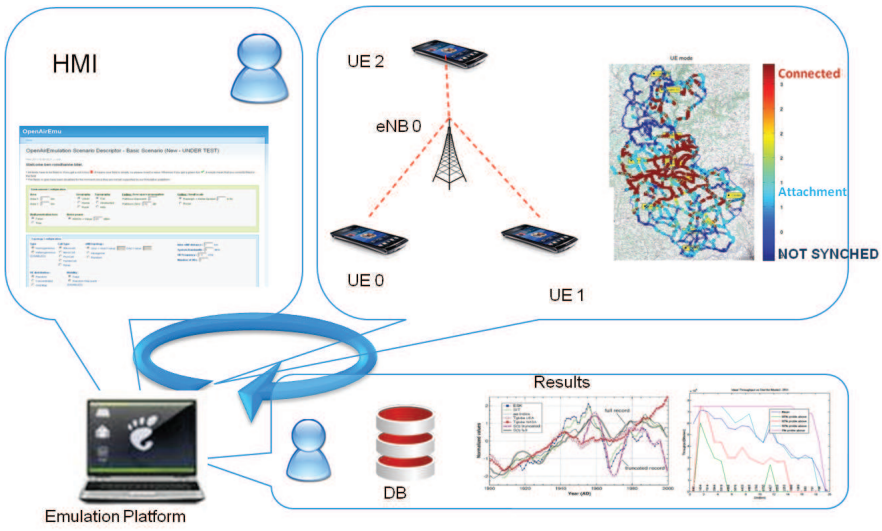


Figure A.2: The user experimentation workflow:

the user' workflow starts from scenario description using an XML file, passing per the simulation achievement until finishing by the output usage.

patibility between new contributions, the integrator decides to resolve the conflict using a predefined correction protocol or to cancel the contribution that makes the problem. In finite, the developer will receive an integration report that include the current state, a list of detect bugs and requirements. We note that this development routine is largely inspired from that of $ns-3$ [69] for the development side and that of *Emulab* [141] for the users' side.

A.4 Conclusion

Experimental methodology is a fundamental corner of any experimentation framework. Thus, addressing the scalability issue of a discrete events simulation requires a trustable methodology that guarantees the reproducibility, the applicability and revisability. In this chapter, we survey OAI experimentation workflow, composed of five consecutive steps: scenario description, configuration, execution, monitoring and analysis. The implementation of that methodology on the OAI platform is splitted on two independent sides: user and developer. Each side defines a specific implementation of that methodology which ensures the correctness and the coherence of the platform and experimentations. Accordingly, the three first requirement of a valid scientific experimentation are guarantees, we can focus on the scalability issue.

Appendix B

Résumé Étendue

Introduction

De nos jours, la notion de réseaux a évolué dans trois sens : la composition, la taille et le trafic. Ainsi, l'importance des équipements mobiles et nomades devint si importante que le nombre de tablettes et de smart phones vendus en 2013 dépassèrent celui des ordinateurs fixes et portables. Le nombre d'équipements a augmenté de sorte que de plus en plus de réseaux privilégient le IPv6 et le trafic mondial atteint une telle densité que plusieurs fibres optiques maritimes ont atteint leur limite.

D'autre part, l'évolution très rapide des technologies sans fils exige un cycle de développement rapide et efficace. Dans ce contexte, la simulation numérique est une étape importante dans l'évolution des systèmes en réseaux et tout particulièrement les réseaux mobiles à grande échelle. En effet, il est important de pouvoir simuler de trés grandes topologies allant jusqu 'à des centaines de millions de noeuds.

La simulation à évènements discrets est tout particuliérement adèquate à la modélisation de l'évolution de tels réseaux. Elle permet notamment de décupler la puissance de calcul disponible en se basant sur des mécanismes de distribution qui font appel à la coopération des processus logiques dans un même scenario.

Cette approche consistait à associer un processus logique avec un processus physique (CPU-core). Cependant, le surplus de latence et de gestion augmente rapidement avec le nombre de noeuds de calcul, présentant de ce fait une limite d'extensibilité. Cette méthode s'avère limitée et ne permet pas d'exploiter les possibilités de calcul parallèle offertes par les nouveaux ordinateurs.

En effet, un ordinateur récent est un système de calcul hétérogène qui embarque des processeurs multi-cores et des processeurs de rendu graphique (GPU) qu'on pourrait utiliser pour des opérations génériques. Dans ce travail, les GPU retiennent toute notre attention pour deux raisons : d'abord, un GPU embarque des centaines de processeurs capables de fonctionner en parallèle. Ensuite, les GPU embarquent une mémoire très rapide en comparaison avec la mémoire principale (RAM). L'utilisation des GPU et des CPU dans une même simulation en parallèle permet de maximiser la puissance de calcul disponible et de profiter d'une bande passante gigantesque.

Ce travail met l'accent sur les avantages de ces ordinateurs et les problématiques relatives à la modification de l'architecture logicielle. Pour ce fait,

nous mettons en évidence l'intérêt de paralléliser la simulation au niveau de l'évènement discret. Nous nous focalisons sur l'ordonnancement des évènements comme un goulot d'étranglement. Nous discutons le concept d'évènements groupés qui se gèrent comme un seul et nous concevons un ordonnanceur hybride qui maximise l'utilisation des CPU et des GPU.
Nous validons l'intérêt de nos propositions avec un simulateur développé pour cette fin, mais aussi avec une version modifiée du simulateur réseaux populaire NS-3.
Le manuscrit est organisé comme suit:
Une introduction générale se focalise sur l'objectif de ce travail et la méthodologie de recherche et d'analyse que nous avons entammé. L'introduction inclut une liste de publications et de délivrables que nous avons réalisé durant les travaux de cette thèse. La suite du manuscrit est organisée en deux parties. La première partie est une étude de l'état de l'art et se compose de quatre chapitres:

- Expérimentation réseaux.
- Simulation à évènement discret.
- Les tendances des nouveaux hardware.
- Etat de l'art de la simulation à grande échelle.

La deuxième partie se compose de quatre chapitres qui mettent en évidence mes contributions:

- Cunetsim: une plateforme expérimentale à la découverte des horizons du passage à léchelle.
- Ordonnanceur d'évènements hybride.
- Architechture logicielle trois-tier: coordinateur-master-travailleur.
- Etude de cas d'une optimisation parallÈle et distribuée: NS-3

Etat de lárt

Expérimentation Réseaux

L'expérimentation est un support de validation primordial et dont l'importance augmente considérablement avec la complexification des réseaux et de leurs technologies et en particulier les réseaux sans fil et mobiles. Dans la littérature scientifique nous identifions cinq catégories d'outils d'expérimentation réseaux:

- Expérimentation sur le terrain: une expérimentation qui se déploie sur un réseau d'exploitation ou un réseau réel qui inclut un trafic réel est dite

sur le terrain. Cet outil d'expérimentation présente l'avantage de produire des résultats réalistique et l'inconvénient d'être non reproductible et compliqué à mettre en oeuvre.

- Expérimentation dans les frameworks réels:Un framework réel est un outil d'expérimentation qui fait appel à des équipement et des logiciels réels. A la différence d'une expérimentation sur le terrain, un framework est complètement dédié à l'expérimentation. Du fait, les expérimentations sont réalistiques et relativement reproductibles. Toute fois, la mise en oeuvre d'une expérimentation dans un framework réel nécessite un investissement logistique considérable.

- Emulation numérique: pour introduire une certaine fléxibilité et agilité dans l'expérimentation, l'emulation se base sur la modélisation d'une partie du système étudié. La partie modélée peut être logicielle ou matérielle. L'utilisation de l'émulation augmente le contrôle et la flexibilité de la mise en place de l'expérimentation mais réduit l'aspect réaliste des résultats obtenues. Il est important de mentionner que l'emulation augmente considérablement la scalabilité de l'expérimentation en terme du nombre d'éléments étudiés.

- Simulation numérique: une simulation numérique consiste à modéliser le système étudié en utilisant une représentation numérique. La simulation peut faire appel à des équations différentielles ou des algorithmes complexes pour représenter le système. Dans les simulations à évènement discret, le temps de la simulation est discret et les composants élémentaires du systme sont représentés par des machines d'états finies. L'avantage de la simulation est de permettre un passage à l'échelle rapide et une expérimentation simplifiée. Toute fois, l'utilisation du modèle pour représenter le système en question, réduit le niveau de certitude des résultats produits.

- Expérimentation hybride: une expérimentation hybride fait appel à plusieurs outils en fonction de la complexité de la partie étudié. Ainsi, une expérimentation à large échelle pourrait simuler le coeur du réseaux, emuler les protocoles réseaux pour les technologies 4G et utiliser un équipement réel pour les transmissions radio. Une expérimentation hybride augmente le niveau du réalisme des résultats concernant les parties importantes du système étudié.

Simulation à évènement Discret

Introduction

Selon les revues de la littérature, de la simulation à événements discrets (DES) est le fait de modéliser le fonctionnement d'un système donné comme une séquence d'événements discrets dans le temps. Chaque événement se produit à un moment-point de l'axe de temps de simulation donné et marque un changement d'état dans le système. Simulation à événements discrets est largement utilisé comme une approche de la conception de logiciel majeur dans la simulation scientifique. La raison derrière le succès de la simulation à base d'événements discrets dans les réseaux informatiques est que le paradigme de simulation s'adapte très bien aux systèmes considérés. En fait, DES fournit un moyen simple et flexible pour réaliser des expériences complexes et d'étudier le comportement des systèmes sous diverses conditions.

Terminologie et Composants

Il existe trois notions partagées entre la majorité du DES : l'entité, le système et le système discret.

- Une entité est une abstraction d'un sujet d'intérêt particulier. Une entité est décrite par ses attributs, par exemple, un paquet de l'entité pourrait avoir des adresses attributs longueur, source et destination. Le terme objet est souvent utilisé comme synonyme.

- Un système est défini par un ensemble d'entités et leurs relations. L'ensemble des entités et leurs relations remplir une fonction, c'est à dire, le système a un certain objectif qu'il cherche à atteindre.

- Un système discret est un système dont l'état, défini par l'état de toutes les entités du système, modifie seulement en des points discrets dans le temps. Le changement de l'état est déclenché par l'apparition d'un événement. Quel événement est exactement, dépend principalement sur "le système et sur "l'objectif de l'étude. , Par exemple, l'envoi et la réception d'un paquet, le déplacement dans l'espace ou la mise à jour de l'état de la batterie.

Habituellement, le système d'intérêt est assez complexe. Afin d'évaluer sa performance par simulation informatique d'un modèle est construit. Le modèle est une représentation logicielle du système, d'où il est constitué d'entités sélectionnées du système d'intérêt et les relations entre les entités sélectionnées. Par convention, le modèle est un système lui-même. Dans les simulations informatiques, il est toujours le modèle qui est considéré, principalement pour réduire la complexité et le coût et les efforts associés.

Le principe

L'idée d'un simulateur à événements discrets est de passer d'un événement à l'autre, de sorte que la survenance d'un événement peut déclencher des

changements dans l'état du système ainsi que la génération d'événements futurs. Les événements sont enregistrés comme cas descripteur (aussi connu comme l'avis de l'événement) dans la future liste d'événements (FEL), qui est une structure de données appropriée, une structure temporelle ordonnée pour gérer tous les événements de la simulation à événements discrets. Un descripteur d'événement est composé d'au moins deux informations (heure, événement) où le temps spécifie le moment où l'événement se produit, et le type donne le type d'événement. L'avenir liste d'événements devrait mettre en 'uvre des fonctions efficaces d'insérer, de trouver et de supprimer les cas descripteur, qui sont placés dans la future liste d'événements. Avec chaque événement ti de temps discret d'un instantané du système est créé dans la mémoire de l'ordinateur qui contient toutes les données nécessaires pour progresser la simulation. En général, tous les simulateurs à événements discrets part les éléments suivants :

- L'état du système : un ensemble de variables qui décrivent l'état du système.
- Horloge : l'horloge donne l'heure au cours de la simulation.
- Liste des événements futurs : une structure de données appropriée pour gérer les événements
- Les compteurs statistiques : un ensemble de variables qui contiennent des renseignements statistiques sur les performances du système.
- Initialisation routine : une routine qui initialise le modèle de simulation et règle l'horloge à 0.
- Routine de distribution : une routine qui récupère le prochain événement de la future liste d'événements et avance l'horloge à l'heure de survenue de l'événement.
- Routine de l'événement: une routine qui est appelée quand un événement particulier se produit au cours de la simulation.

Simulation à événements discrets parallèle

Depuis simulation à événements discrets a été adopté par une large communauté de recherche, les développeurs de simulation ont tenté de tirer les bénéfices de l'exécution d'une simulation sur plusieurs unités de traitement en parallèle. Ainsi, un large éventail de recherches ont été menées sur la simulation de l'événement parallèle discrets (PDES). Dans le reste de cette section, nous passons en revue les défis de parallèle DES l' architecture commune et les principaux algorithmes de synchronisations.

Discrets limites de la simulation de l'événement

Les systèmes technologiques sont de plus en plus complexe avec l'émergence de nouvelles technologies qui combinent large interconnexion avec les structures et architectures composites. En général, deux tendances orthogonales en termes de complexité peuvent être identifiés : (i) une augmentation de la complexité structurelle et (ii) une augmentation de la complexité algorithmique. Tant imposer des exigences élevées sur l'architecture de simulation et le matériel exécution des simulations.

On note la taille d'un système simulé comme un indicateur de la complexité de la structure d'un modèle de simulation. En ce qui concerne la recherche de réseau, systèmes de culture, comme les réseaux peer-to -peer et les réseaux mobiles omniprésents, induit une augmentation considérable de la taille des systèmes de communication. Ces grands systèmes ajoutent généralement des caractéristiques complexes qui ne peuvent être observés dans les réseaux de plus petite taille (par exemple, les bancs d'essai) ou capturés par des modèles analytiques. Ainsi, afin d'étudier les caractéristiques, les modèles de simulation d'un grand nombre de noeuds de réseau simulées. étant donné que chaque noeud de réseau est représentée en mémoire et déclenche des événements dans le modèle de simulation , la consommation de mémoire et l'augmentation du temps de calcul considérablement . Même si le réseau étudié est relativement faible, la complexité de calcul devient un facteur important si le modèle de simulation est très détaillé et implique de nombreux calculs. En particulier, les réseaux sans fil qui utilisent des technologies radio avancées telles que OFDM (A) et les Turbo Codes tombent dans cette catégorie . Des modèles sophistiqués de propagation radio, la modélisation d'interférence, et les modèles de codage du signal augmentent la complexité de l'ensemble.

Cadres de simulation visent à compenser ces problèmes en permettant des simulations pour être exécutés parallèlement sur plusieurs unités de traitement. En combinant les ressources de mémoire et de calcul de plusieurs unités de traitement, le temps de simulation peut être réduite au coût des besoins de mémoire plus élevés et les frais de gestion. Bien que cette approche soit connue depuis plus de deux décennies, les récents progrès technologiques permettent de réduire considérablement le coût du matériel de l'infrastructure de calcul parallèle. Rendant ce matériel à la disposition de la communauté de recherche à grande qui la main sur le feu des projecteurs sur la discipline.

Principes de la simulation à événements discrets parallèle

L'approche adoptée par le PDES est de diviser un modèle de simulation dans plusieurs instances qui sont exécutés sur les unités de traitement indépendantes en parallèle. Le défi central de PDES est ainsi à maintenir l'exactitude de la simulation. Tout cadre simulation présente trois centrales structures de données : i) les variables d'état du modèle de simulation, ii) une liste horodatée des événements, et iii) une horloge mondiale. Au cours d'une simulation , le planificateur supprime en permanence l'événement avec le plus petit times-

tamp (emin) de la future liste d'événements (FEL) et exécute la fonction associé . T désigne la fonction d'horodatage qui attribue une valeur de temps de chaque événement et E est l'ensemble de tous les événements dans la liste des événements. Bien que la fonction de gestionnaire soit en cours d'exécution, les événements peuvent être ajoutés ou supprimés de la liste d'événements. Choisir emin est cruciale car sinon la fonction de gestionnaire d'un ex événement avec T (emin) < T (ex) pourrait changer les variables d'état qui sont ensuite accessibles quand emin est manipulé. Dans ce cas, l'avenir (ex) aurait changé le passé (emin) que nous appelons une violation de causalité . Ainsi, nous formulons le défi central de PDES comme suit : étant donné deux événements E1 et E2, décider si les deux événements ne gênent pas, permettant ainsi une exécution simultanée, ou pas , donc nécessitant une exécution séquentielle . Simulation des cadres de travail parallèles emploient une grande variété d'algorithmes de synchronisation de trancher cette question. La section suivante présente une sélection d'algorithmes fondamentaux et discute leurs propriétés.

Modèle De Simulation Parallèlement Et Algorithmes

Un modèle de simulation parallèle se compose d'un nombre fini de partitions qui sont créées conformément à un schéma de partitionnement spécifique. Trois exemplaires régimes de partitionnement sont i) régime espace parallèle de partitionnement, ii) canaliser partitionnement parallèle, et iii) le temps partitionnement parallèle. Le schéma de partitionnement espace parallèle divise le modèle de simulation ainsi que les connexions entre les n'uds simulés. Par conséquent, les cloisons résultant constituent des groupes de noeuds. Les bases du régime partitionné canal parallèles sur l'hypothèse que les transmissions qui utilisent différents (radio) des canaux, les médiums, les codages etc. n'interfèrent pas. Ainsi, les événements sur les n'uds non interférents sont considérés comme indépendants. Par conséquent, le modèle de simulation est décomposé en groupes de noeuds non interférents. Cependant, parallèlement canal partitionnement n'est généralement pas applicable à tous les modèles de simulation, laissant ainsi pour spécialisées scénarios simulation. Enfin, les dispositifs de séparation parallèles de temps subdiviser le temps de simulation d'un essai de simulation dans des intervalles de temps de même taille. La simulation de chaque intervalle est considérée comme indépendante des autres sous l'hypothèse que l'état du modèle de simulation est connu au début de chaque intervalle. Cependant, l'état d'une simulation de réseau comprend généralement une complexité importante et n'est pas connue à l'avance.

Tendances de l'évolution du matériel

Introduction

Dans ce paragraphe nous visons à mettre en évidence l'évolution du matériel qui s'est accumulés au cours des dernières décennies. En particulier, nous examinons le parallélisme et l'optimisation des caractéristiques qui concernent les processeurs et l'émergence de coprocesseur spécialisé que la démocratisation grande échelle du calcul parallèle. Dans ce contexte, nous présentons le GPU comme une puce massivement multi-core et l'accélérateur de l'informatique comme un dispositif multi-core dédié. Dans le reste de ce chapitre, nous examinons d'abord l'architecture CPU. Deuxièmement, nous présentons un apercu de l'état logique de l'art de GPU. Troisièmement, nous résumons le concept d'accélérateur multi-core qui a été inauguré par le processeur Xeon Phi. Enfin, nous concluons ce chapitre par une enquête auprès des API de programmation parallèle pour ce type de matériel.

Evolution de l'informatique Chips

CPU: Evolution historique et tendances

Historiquement, la CPU a été conÃ§u pour réaliser toutes les tâches informatiques demandés par le système d'exploitation. Depuis l'introduction du micro-processeur 8086 et la démocratisation de l'ordinateur, les améliorations de la CPU disent deux routes: l'augmentation de la fréquence du CPU et s'étendant aux instructions énoncées. Néanmoins, l'augmentation de la fréquence de la puce informatique induit une augmentation phénoménale dans les paramètres thermiques et de consommation d'énergie. Par conséquent, au début de la décennie précédente, les fabriquants de CPU reconnaissent que l'augmentation de la fréquence ne peut pas être la future approche de développement. Les tendances communes ont été de juxtaposer deux (ou plus) processeurs sur la même puce. Les premières tentatives étaient vraiment la juxtaposition de deux processeurs indépendants qui limite toute collaborations avancées et sature le bus interne. Pro progressivement, de l'architecture du processeur évolue et devient composée de plusieurs noyaux computing et stades de mémoire sur puce. La communication entre les différents cores est progressivement optimisée, et la gestion de la mémoire distribuée devient mature. Cependant, le concept de multi-core CPU repose sur l'indépendance de chaque noyau de sorte que chacun d'eux est capable d'exécuter n'importe quelle tâche.

Par conséquent, tous les noyaux sont construits suivant la même architecture complexe qui, en retour, de réduire la capacité de maximiser le nombre de cores intégrés. Au meilleur de notre connaissance, le plus grand processeur multi-core comprend jusqu'à 16 cores, qui est une très petite échelle par rapport aux GPU que nous présentons plus loin. De l'autre côté, l'amélioration des instructions établies demeure pertinente. En particulier, l'introduction d'instructions vectorielles est pertinente pour les usages scientifiques et mul-

timédias. Ces instructions permettent le traitement de plusieurs mots en un seul cycle d'horloge. Dans l'état actuel de la technique, il est possible de traiter jusqu'à quatre mots (256 bits) en parallèle par chaque noyau en utilisant le jeu d'instructions AVX. AVX2 promet le traitement de 8 mots dans la prochaine génération de CPU.

En parallèle, les fabricants de CPU de continuer à améliorer l'architecture du processeur à pro-Vide fonctionnalités de plus en plus. La tendance est que, entre les deux générations successives, le gain est compris entre 10% et 20%. Une caractéristique supplémentaire pertinente est le multithreading simultané (SMT), noté que la technologie Hyper-Threading d'Intel et partiellement mis en oeuvre par AMD. Le concept repose sur une planification d'instructions de pointe qui permet à chaque core l'exécution de deux (ou plusieurs) des fils. Chacune d'entre elles est supposée être exécutée sur un noyau virtuel. En fonction de l'optimisation du logiciel, le gain relatif est compris entre 10% et 25%. Pour conclure, la combinaison de ces éléments permet au processeur d'être une solution compétitive. Cependant, l'architecture globale de l'ordinateur est construite autour d'un petit nombre de puces qui partagent le même matériel (entre un et quatre puces). Dans le même temps, le développement de supports de communication évolue différemment: dans la communication fournit une bande passante importante tandis que la communication à l'extérieur de la machine est encore lent. Cette restriction particulière favorise l'utilisation de coprocesseurs que nous allons détailler dans les sections suivantes.

GPU: Evolution historique et tendances

Par définition, une unité de traitement graphique (GPU) est un coprocesseur qui assure le rendu graphique. L'évolution du processeur graphique actuelle commence avec l'introduction des premiers appareils 3D. Les premiers systèmes graphiques sélectionnés d'un pipeline de fonction fixe (FFP), et l'architecture suivant un trajet de traitement très rigide en utilisant presque autant d'API graphiques comme il y avait des fabricants 3D. Alors que la période des années 90 comptait de nombreux GPU innovante fabricants, le début de 2000 a été marquée par la concentration de l'industrie du GPU autour de deux acteurs: NVIDIA et ATI. La principale évolution de ce temps a été l'introduction de plus en plus souple API de programmation tels que le DX7 et l'OpenGL (1999-2000) et le support de 32 bits à virgule flottante informatique avec GPU. Une révolution dans l'industrie du GPU a été le début des premiers graphiques unifiés et GPU Computing, programmé en C avec CUDA. La Geforce 8800 était le premier GPU compatible CUDA qui comprend jusqu'à 128 cores CUDA alors que le CPU le plus puissant de l'époque comprend deux noyaux. Le développement de GPU a augmenté dans deux directions: la simplification de l'interface programmation et l'augmentation du nombre de core de calcul. Ainsi, l'API OpenCL est principalement soutenu par la majorité des fabricants de puces, notamment AMD, NVIDIA, Intel et

ARM. Il est pertinent de souligner que les GPU actuels sont des dispositifs orientés débit-composées de centaines de cours de traitement. Ils maintiennent un débit élevé et une mémoire cache de latence par multithreading entre des milliers de threads. GPU reposent sur une architecture hiérarchique à deux niveaux. Il est composé de vecteur pro-processeurs au plus haut niveau, appelé multiprocesseurs (SMS) en streaming pour les GPU NVIDIA et SIMD noyaux pour les GPU AMD. Chaque processeur vectoriel contient un tableau des cors de traitement, appelé processeurs scalaires (PS) pour les GPU NVIDIA et de l'unité de traitement de flux pour GPU AMD. Tous les cores de traitement à l'intérieur d'un processeur vectoriel peuvent communiquer à travers un mémoire gérés par l'utilisateur sur puce, appelée mémoire partagée pour les GPU NVIDIA et la mémoire locale pour les GPU AMD. Le CUDA [148] et [133] OpenCL API partagent la même SPMD (Programme Single Multiple Data) modèle de programmation. CUDA vitalise SMS blocs (équivalent aux groupes de travail en OpenCL) et SPS fils (équivalents à des éléments de travail dans OpenCL), qui permettent aux programmeurs d'exécuter des milliers de threads et de blocs à travers les différentes générations de GPU indépendamment de la quantité de processeurs physiques.

Programmation parallèle: Modèles et API

Le but du calcul parallèle est d'améliorer les performances en exécutant l'application sur plusieurs processeurs. Même si le calcul parallèle est traditionnellement corrélé avec la communauté HPC, il est de plus en plus commun pour intégrer l'informatique en raison de l'émergence récente de l'architecture multi-core. Cependant, l'utilisation optimale de ce matériel nécessite des outils de programmation adéquate qui optimisent les fonctionnalités de parallélisme. En ce sens, il existe plusieurs API qui fournissent un appui de la programmation distribuée parallèle et. En particulier, nous identifions cinq API représentatives: les Pthreads, OpenMP, MPI, l'OpenCL et CUDA.

Pthreads

Pthreads est l'acronyme de l'interface du système d'exploitation (POSIX) Portable Threads. Pthreads sont mises en oeuvre dans un en-tête (pthread.h) et une bibliothèque pour la création et la manipulation de chacun des travailleurs appelés threads. Travailleur administration dans Pthreads nécessite de gérer explicitement le fils du cycle de vie de la création à la sortie. En outre, la définition de la division de la charge de travail et la cartographie doit être explicitement défini par le concepteur de logiciels. Afin de préserver section critique, c'est à dire la partie du code qui accède à des données partagées, Pthreads accorde mutex (exclusion mutuelle) et sémaphore. Mutex autorise un seul thread d'accéder à une section critique à un moment donné, alors que sémaphore permet différents threads d'accéder à une section critique.

OpenMP

OpenMP est une spéciation générique qui considère la mémoire partagée paral-

lélisme. Il se compose d'un ensemble de routines de bibliothèque d'exécution et les variables d'environnement qui simplifient la programmation parallèle dans un contexte de mémoire partagée. OpenMP est disponible comme une extension des programmes Fortran, C et C + +. Le travailleur primaire de la conception OpenMP est fils. La gestion des travailleurs est implicite et repose sur l'utilisation de pré-compilation directive Pragma qui indiquent qu'une section donnée peut être exécutée en parallèle. Le nombre de fils parallèles est une variable environnementale qui dépend des capacités matérielles. Ainsi, contrairement à Pthread, l'implication de révélateur est limitée à la conception des fils interaction et l'utilisation des données. Charge de travail partitionnement et la cartographie tâche à travailleur exigent relativement peu d'effort de programmation. OpenMP aussi abstraction de la distribution de la charge de travail entre les travailleurs et la faÃ§on dont les tâches sont les filets.

MPI

MessagePassing Interface (MPI) est une spécification pour la transmission de messages opérations. Le concept de MPI estime que chaque travailleur est un processus indépendant. MPI est actuellement le standard pour le développement d'applications HPC sur l'architecture de mémoire distribuée. Il fournit des liaisons de langage pour C, C + + et Fortran. Certains des implémentations les plus populaires MPI comprennent OpenMPI, MVAPICH, MPICH, GridMPI, et LAM / MPI. Le contrÃ´le des travailleurs est assuré à l'aide d'un outil de ligne de commande (ou des scripts) et le système assure la gestion des différents processus entre le matériel disponible selon la configuration donnée. Les programmeurs doivent contrÃ´ler ce que les tâches doivent être calculés par chaque processus. Communication entre les processus adopte le paradigme message passe. MPI classe largement ses opérations de transmission de messages de point-à-point et collective. MPI Barrier est utilisé pour spécifier que la synchronisation est nécessaire. Les blocs de fonctionnement de la barrière jusqu'à ce que tous les processus en cours atteint par tous les processus qui participent à la barrière.

CUDA

Le Compute Unified Device Architecture (CUDA) est un modèle de programmation à usage général pour écrire des applications hautement parallèles. Il fournit plusieurs abstractions clés: une hiérarchie de blocs de filetage, mémoire partagée, et la synchronisation de barrière. Dans l'état actuel de la technique, CUDA est exclusivement compatible avec les GPU Nvidia. Le modèle considère un système parallèle d'un couple d'un hÃ´te (c.-à-CPU) et les périphériques (c.-à-GPU). Tâches de calcul parallèles sont effectuées dans GPU par un ensemble de threads parallèles. Le modèle organise les discussions sur une hiérarchie à deux niveaux, à savoir le bloc et la grille. Block est un jeu de fils clonés, chacun d'eux est identifié par un Tid (fil id), tandis que la grille est un ensemble de couplage lâche de blocs. La gestion des travailleurs est

implicitement réalisée par le pilote CUDA. Ainsi, les programmeurs à spécifier les paramètres de la grille et le bloc requis pour traiter le travail donné. Il doit être mentionné que la synchronisation des threads est fait implicitement en utilisant des routines disponibles tels que la de fonction syncthreads.

OpenCL

OpenCL est une nouvelle norme pour le travail parallèle et informatique hétérogène de données parallèle sur une gamme de processeurs modernes, GPU, DSP, et d'autres conceptions microprocesseur. OpenCL fournit des abstractions et un ensemble d'API de programmation basé sur les réussites passées avec CUDA, TBB, et d'autres outils de programmation. OpenCL définit un ensemble de fonctionnalités de base qui est pris en charge par tous les appareils, ainsi que des fonctionnalités en option qui ne peut être mis en oeuvre sur des dispositifs à haute fonctionnalité, et comprend un mécanisme d'extension qui permet aux fournisseurs d'exposer des caractéristiques matérielles spécifiques et des interfaces de programmation expérimentaux pour le bénéfice des développeurs de logiciels. Bien que OpenCL peut pas masquer des différences importantes dans les architectures matérielles, il ne garantit pas la portabilité et l'exactitude. Cela le rend beaucoup plus facile pour un développeur pour commencer un programme OpenCL fonctionne correctement réglé pour une architecture, et de créer un programme de fonctionnement correctement optimisé pour une autre architecture.

Les travaux liés

Introduction

Dans ce chapitre, nous proposons de mettre en évidence les travaux connexes qui répondent aux questions de simulation parallèle et distribué sur les ordinateurs hétérogènes. Le chapitre est composé de trois sections. La première section présente les approches les plus courantes qui traitent simulation à grande échelle. La deuxième section met en lumière les problèmes de planification de l'événement et la troisième section résume optimisations les plus courants qui émergent dans le domaine de la simulation distribuée.

Simulation à grande échelle

Dans les revues de la littérature, il existe trois principales approches pour faire face à la simulation des grandes échelle: (1) parallèles à base de CPU et simulation distribuée, (2) l'accélération partielle à l'aide spécifique co-processeur et (3) l'approche entièrement GPU. Une.

- En parallèle sur la base d'une unité centrale de traitement et la simulation distribuée [98], la plate-forme est composée de plusieurs instances de simulation qui collaborent pour assurer la simulation d'une manière donnée. Le plus commun distribué et parallélisation de la simulation basée sur l'UC sont:

  - Répartition spatiale: chaque instance simule une partie de l'espace et / ou de la population mondiale.
  - La répartition fonctionnelle: chaque instance assure un (s) tâche / fonc-tion (s) pour la simulation globale.

  La distribution de collaboration: comprend simulations distribuées dynamiques et la simulation qui combine l'utilisation de plusieurs cours de processeur par cas de simulation. Dans la terminologie la plus courante, nous parlons d'une approche fédérée pour résumer cette catégorie. Une telle approche fédérée utilise des modèles existants et fournit une parallélisation rapide de simulateurs séquentiels existants. Dans ce contexte, GTNetS est un cadre expérimental qui démocratise l'utilisation de l'architecture plat / hiérarchique pour gérer à grande échelle réseau sim-lation. Ensuite, l'expérience de développement de GTNetS a été appliquée aux open source ns simulateur de réseau - 3 avec une bonne performance en terme de évolutivité. Cependant, cette approche induit une perte en raison de la synchronisation entre les différents processus et / ou des machines et nécessite simulation sophistiqués et coûteux en infrastructure. Cette surcharge peut augmenter considérablement dans un environnement mobile si la topologie du réseau et de la cartographie de la machine n'est pas dynamiquement géré (par exemple grâce à la migration de noeud). Pour la majorité des simulateurs à base de CPU, la

dégradation des performances se produit lorsque la simulation combine les facteurs limitants tels que le taux de mobilité, le nombre de nouds, et augmentation de la charge de trafic. En ce qui concerne les simulateurs distribués, comme la dégradation des performances se produit lorsque les inter-machines de communica-tion augmente. Une démonstration de l'évolutivité, sur la base de la NS-3 distribué a soigneusement évité le problème de l'interaction entre des noeuds dans différentes machines de simulation. Même si simulateurs parallèles et distribués ont dépassé une limite d'évolutivité, ils introduisent de nouveaux pro-blèmes tels que le coût d'un noeud simulé, la stratégie de distribution des nouds initial et leur migration dans les différentes machines.

- L'accélération partielle vise à accroitre l'efficacité de la simulation au niveau local en déchargeant la partie la plus intensive du processeur d'une simu-lation donnée de la CPU à un co-processeur dédié. Le FPGA a été largement utilisé comme une solution d'accélération [30] Toutefois, dans certaines approches récentes, le GPU est utilisé pour décharger des tâches de calcul intensif telles que la modélisation du canal [10] et les files d'attente [102] dans le simulateur. Des études récentes recommandent l'utilisation du GPU pour plus simulation à usage général [110], ou même comme une architecture de simulation accélération GPU lorsque la précision et les performances d'exécution sont à la fois critique [7]. Ainsi, le GPU devient une alternative de plus en plus attrayant pour la base de CPU-par-allélisme cher, avec une puissance de calcul importante à un coût relativement faible. Avec l'avènement de la GPU de la série GeForce8 en 2006 et le calcul architecture de dispositif unifié (CUDA) [95], le GPU devient un support informatique disponibles. Même si cette approche réduit considérablement le temps de calcul, la simulation reste principalement dans la CPU qui continue d'être le goulot d'étranglement du système dans les grandes scénarios d'échelle. En outre, un transfert continu de données entre la mémoire de GPU et le CPU d'une présente une sérieuse limitation.

- L'approche entièrement GPU vise à décharger la totalité des tâches de simulation sur l'espace de GPU. En outre, la mémoire nécessaire sera exclusivement sur le GRAM. Cette approche offre trois avantages:

  - L'impact de la CPU est réduite au minimum, en contraste avec la simulation parallèle traditionnelle.
  - Le transfert de la mémoire entre la mémoire principale et le GRAM est pratiquement nulle au cours de la simulation.
  - Le temps de latence de synchronisation entre les différents noyaux de la simulation est réduite.

Cependant, le GPU n'est pas entièrement conforme X86, ne supporte pas les caractéristiques du CPU, a besoin d'une architecture logicielle particulière à divulguer sa puissance et ne prend pas en charge le mécanisme de verrouillage de la mémoire. En raison de ces contraintes, l'approche de simulation entièrement GPU est peu étudiée. Dans le même contexte, nous proposons une preuve de concept, noté Cunetsim qui utilise le GPU comme un environnement de simulation principal et le CPU comme contrôleur. Cunetsim est une plate-forme de simulation expérimentale permettant la validation et l'expérimentation de nouvelles approches. Contrairement aux travaux précédents, Cunetsim est concu pour fournir un environnement exécu-tion parallèle indépendant pour chaque noeud simulé. Noeuds communiquent par envoi de messages basé sur l'échange de tampon. Ainsi, le cadre permet d'éviter l'utilisation de la connaissance mondiale et augmente le niveau de parallélisme. Il est basé sur le modèle maître / travailleur pour une co-simulation CPU-GPU et fournit le modèle de synchronisation hybride qui maximise l'efficacité et garantit l'exactitude de la simulation. La simulation exploite le grand nombre de cours de calcul du GPU pour exécuter nouds en parallèle et l'accès à la mémoire à grande vitesse pour réduire noeud de communication de latence.

Cunetsim: Une platforme d'experimentation à la decouverte du potentiel du parallel

Simulateurs de réseau au niveau des paquets sont généralement basées sur un paradigme à événements discrets où le système simulé est le modèle en utilisant une séquence d'événements . Chaque événement a un horodatage discrets qui définissent un ordre strict de l'exécution . Dans la simulation à événements discrets , chaque événement est représenté par un descripteur qui inclut l'horodatage et des paramètres d'exécution (soit un rappel) . Un cadre simulation comprend un planificateur d'événement qui gère l'événement descripteur et définit l' ordre d'exécution des événements . En général , ces événements représentent mobilité , la connectivité , la modélisation moyenne (filaire ou sans fil) et dans / sur le traitement des paquets . La complexité du temps et de la mémoire utilisée d'une simulation donnée sont proportionnelles au nombre d' événements . Néanmoins, le nombre d'événements gen - survoltage augmente de faÃ§on exponentielle en fonction à la fois du nombre total de noeuds et la charge de trafic . En conséquence , la gestion de cas devient le principal goulot d'étranglement lors du ciblage de simulation à grande échelle . Il y a aussi un compromis entre la précision des modèles , modèles en particulier air - moyennes (propagation des ondes) , et de la complexité de temps qui doit être pris en compte lors du ciblage de simulation à grande échelle .

En conséquence , l'exécution parallèle et distribuée apparaît un candidat trivial car il fournit plus de puissance de calcul et de mémoire. Cependant , une telle approche présente de nouveaux défis dans le niveau de gestion . Dans la littérature re- vues , nous identifions que la communication entre les différents disques de simulation est particulièrement coûteux [48] . En outre, nous notons que la majorité des cadres existants sont basés sur une approche distribuée qui ignorent les nouvelles fonctionnalités matérielles , malgré le fait que l'accélération potentielle de la simulation parallèle est clairement identifié et prouvé dans plusieurs publications pionnières . Travaux de recherche d'enquête qui attirent la limitation de la simulation parallèle dans un con - texte , identifier la programmation de l'événement comme un obstacle majeur qui ne peut être évité . En effet, même s'il est possible d'exécuter simultanément plusieurs événements , le processus de programmation reste centralisée qui génère le goulot d'étranglement de simulation parallèle . Plusieurs travaux d'optimisation proposent des approches innovantes.

Cependant, les obstacles inhérents au concept de DES nécessite le passage à travers un chemin d'accès central pour tous les événements . Contrairement aux approches existantes , nous proposons (1) pour générer des événements parallèles plutôt que de détecter leur éventuelle exécution simultanée et (2) pour planifier des événements parallèles clonés qu'une seule fois. Ces deux modifications conceptuelles sont la principale contribution de ce chapitre

. La seconde innovation que nous vous présentons est de savoir comment nous prévoyons de mettre en oeuvre ces concepts . En fait , nous proposons d'utiliser le GPU comme le support de simulation principale tandis que le CPU agit comme le maître de la simulation . Nous visons à utiliser les GPU NVIDIA CUDA basé sur le riche écosystème de développement autour . Nous définissons le modèle de simulation selon quatre points:

- Le composant unitaire de la simulation est le noeud , toute entité simulée est obligatoire par rapport à un noeud donné .
- Le générateur d'événement génère le même événement pour tous les noeuds . Cela signifie qu'à un moment donné , tous les n?uds seront exécutés le même événement .
- Chaque noeud sera exécuté dans un GPU de base indépendante . Ainsi , nous garantissons une exécution simultanée .
- Le descripteur d'événement n'est pas par rapport à un noeud , mais à tous les noeuds . En conséquence , le planificateur d'événements manipuler une entrée .

Nous nous attendons à ce que ce modèle permettra de surmonter les limites de DES par rapport à la programmation traditionnelle . Comme une preuve de concept , nous proposons un nouveau cadre CPU - GPU de co-simulation notée Cunetsim , CUDA Network Simulator . Cunetsim est une plate-forme de simulation expérimentale permettant la validation et l'évaluation rapide . Contrairement aux travaux précédents , Cunetsim est concu pour fournir un environnement d'exécution parallèle indépendant pour chaque noeud simulé. Noeuds communiquent à travers le passage de message basé sur l'échange de tampon . Ainsi , le cadre permet d'éviter l'utilisation de la connaissance mondiale et augmente le niveau de parallélisme . Il est basé sur le modèle maître / travailleur pour une co-simulation CPU - GPU et fournit le modèle de synchronisation hybride qui maximise l'efficacité et garantit l'exactitude de la simulation .

Concepts fondamentaux

Le but de cunetsim est de réaliser une simulation à grande échelle aussi vite que possible . L'idée de l'exécution de la simulation sur le GPU semble prometteur , mais l'utilisation d'une telle puissance de calcul reste délicate en raison de la conception du logiciel requis qui diffère du code x86 traditionnelle . Pour harmoniser les GPU exi-gences avec une spécificité de simulation de réseau , cadre cunetsim s'articule autour de trois concepts fondamentaux : la piscine des travailleurs , la séparation entre l'événement et le descripteur d'événement et le massif génération des événements parallèles .

Le groupe de travailleurs Dans le modèle maître / travailleur un travailleur est assimilé à être un LP qui gère une partie de la simulation .

En général, le travailleur est toujours dans la même machine au cours de l' exécution . Dans le GPU , il est impossible d'affecter un processus à chaque noyau de calcul et le nourrir par les événements. Ainsi, nous considérons que le travailleur représente une entité simulée primaire (définie par des données + + processus FSM) . Les travailleurs mettent modèles le nombre total d' entités simulées et de partager des ressources disponibles .

La séparation entre l'événement et sa description

En DES programmation de l'événement parallèle est un défi permanent qui est largement adressé dans les revues de la littérature [45 , 83 , 128] . Sous des conditions de grande envergure de la programmation de l'événement devient l'un des principaux goulets d'étranglement . De plus , la détection d'événements indépendance afin de les programmer en parallèle nécessite des algorithmes sophistiquées . Néanmoins , quel que soit l'algorithme utilisé pour réaliser la programmation de l'événement, son coût reste proportionnelle au nombre d'évènements . En conséquence , nous avons eu l'idée de dissocier le cas de l'événement de- scripteur , comme le planificateur d'événements gère descripteurs uniques . L'avantage principal de cette approche est la dissociation entre le nombre de cas et de la complexité de la programmation . En fait , si nous réussissons à surcharger le descripteur d'événement afin de représenter plusieurs événements plutôt qu'un seul , alors nous pouvons contourner le goulot d'étranglement de la programmation .

Génération d'événements massivement parallèles

Le concept de parallélisme massif est un modèle de logiciel adapté pour SIMD hardware et en particulier pour la programmation de GPU . L'idée principale consiste à générer fils clonés , dont chacun effectue la même opération sur un ensemble de données indépendantes . Ce concept est dérivé du logiciel de traitement graphique , dans lequel chaque pixel ou d'un polygone sont traitées de facon indépendante et en parallèle par le même algorithme . Nous vous proposons de générer des événements clones plutôt que la détection d'événements qui peuvent potentiellement être exécutées en parallèle . Notre compréhension de ce concept est le suivant: les entités simulées identiques peuvent générer le même événement pour être exécuté dans le même horodatage. Chaque événement est relatif à une entité et sera exécuté sur les données correspondantes, les attributs et la mémoire. Cependant, avec la représentation adéquate des données, il est possible de représenter ces événements avec une entrée sur le système d'ordonnancement. Ainsi, le planificateur d'événements gère une entrée tandis que les ressources d'exécution exécutent plusieurs événements (tel que défini par le générateur).

Scheduler hybride Evénements

Simulation à événements discrets (DES) est largement utilisé pour modéliser , analyser et évaluer des systèmes complexes , où l'analyse formelle est difficile ou non - déterministe . Cependant , l'évolutivité de DES reste difficile en raison du système et la complexité croissantes de modèle , d'une part , et les caractéristiques phénoménales inter- connectivité des systèmes ré- cent , d'autre part . En outre, une limitation fondamentale du courant DES est l'absence d'une politique de gestion de l'événement dédié qui estime les capacités de calcul hétérogènes . Dans ce contexte , la planification de l'événement est identifié comme un goulot d'étranglement inhérent de DES. En particulier, la programmation de l'exécution d'événements à venir , tout en maintenant une charge en continu dans des conditions de grande envergure augmente le coût de la programmation, jusqu'à ce qu'il devienne le goulot d'étranglement [47] . La plupart des approches de programmation recherchées reposent sur un modèle d'ordonnancement d'événements centralisé optimisé principalement pour l'architecture du noeud de calcul homogène . Un tel modèle reste limitée et ne pas exploiter le plein potentiel du matériel moderne . Par conséquent, les approches de planification parallèles et distribués réapparaissent comme un facteur important pour augmenter l'évolutivité sur les architectures informatiques hétérogènes [136] . L'objectif est d'exploiter une mul -titude de processeurs parallèles et interactives unifiées au niveau de l'événement planificateur de coopérer les uns avec les autres. Les exemples incluent les processeurs multi -core , multi- GPU , système - sur - puce multi - processeur et l'unité de traitement accéléré . En ce qui concerne cette nouvelle exigence , nous soulignons la nécessité d'un cadre de planification garantie qui combine abstraction matérielle avec une gestion simplifiée .

La plupart de ces architectures semblent prometteuses , mais leurs écosystèmes dans certains cas, soit ne sont pas pleinement développés ou contradictoires [21] . Avantages de supercomput -teurs de GPU ont été mis en évidence dans [109] , où les auteurs suggèrent de revoir et d'élargir la vision du DES . Néanmoins , la plupart des tentatives récentes supposent la compatibilité ascendante avec le concept d'ordonnancement séquentiel [100 , 140] . Cette méthodologie présente une faiblesse conceptuelle car il considère un noud de calcul multi-core comme une simple extension d'un mono -core un . En outre , pour rester en arrière compat - ible , le gain attendu sera considérablement réduite par rapport à une conception de logiciel dédié qui exploite les capacités de calcul parallèle du matériel en cours ainsi que le temps de latence de communication [4] .

Dans ce travail , nous introduisons un nouveau planificateur d'événement parallèle pour les architectures de com - puting hétérogènes , désigné comme programmateur hybride (H -scheduler) . Le H - programmateur est concu pour allouer dynamiquement des événements de ressources de calcul disponibles

tout en gardant un taux d'événements stable . Ceci est réalisé comme le planificateur est conscient du tas.
de processeurs et de leurs capacités et dispose d'un accès permanent à leurs charges instantanées et le temps d'exécution à travers un mécanisme de rétroaction. L'ordonnanceur fonctionne sur toutes les ressources informatiques disponibles dans le même espace d' adressage mémoire. Pour augmenter l'efficacité de l'ordonnanceur , chaque événement est associé à un descripteur spécifique qui sera stocké dans le 3 - D de la structure de données un ensemble de données en 3 dimensions d'Al- bas du cadre pour faire face aux entrées consécutives (D 1) , presque parallèle les entrées (D 2) et les entrées clonés (D 3) . Puisque l'objectif de ce travail est de maximiser l' efficacité de la simulation , la première tentative a été d'utiliser des stratégies de planification opportunistes . Cependant , nous décidons d'utiliser la politique de planification prudente pour éviter la sur- charge générée par le mécanisme de récupération et le vecteur d'état lors de l'examen politique optimiste dans les milieux parallèles et hétérogènes. Le H -scheduler est composé de quatre principaux processus : Répartiteur d'événement , événement d'injection -teur , GPU - programmateur, et CPU -scheduler , où les événements sont fluides.

- Le répartiteur a extrait les événements nouvellement générés à partir de différentes files d'attente et les ajoute à une position correspondante dans une structure en 3 dimensions données optimisé pour l' exécution en parallèle .

- L'injecteur dirige un groupe d'événements parallèles à la sous- ordonnanceur plus adéquate (CPU ou GPU) en fonction des informations de retour recues .

- Le GPU - ordonnanceur assure l'exécution des entrées groupés sur le GPU dédié .

- Le CPU - ordonnanceur assure l'exécution de toute inscription en marche la CPU dédiée .

Chaque sous- ordonnanceur est optimisée pour un matériel spécifique , afin de maximiser le débit des ressources de calcul correspondant à l'activité . Plusieurs optimisations sont proposées pour accélérer la décision de planification comme le goulot d'étranglement peut changer au fil du temps . Le mécanisme H -scheduler repose sur trois stratégies pour le répartiteur et les processus d'injection : rapides , avancées et hybrides . La politique rapide a pour but de minimiser le coût d' une prise alors que la pointe a pour but d' optimiser la cible d'exécution selon la charge matérielle . La politique hybride utilise des méthodes avancées à la fois rapides et à maximiser la stabilité du système . Les évaluations comparatives ont démontré que le gain de perfor-

mance peut être augmentée par un facteur de 2 par rapport à des planificateurs centralisés et conservateurs.

Bibliography

[1] http://disco.ethz.ch/projects/sinalgo/. 48

[2] http://www.nsnam.org/. 48

[3] http://www.pgroup.com/resources/unifiedbinary.htm/. 48

[4] Brandon G Aaby, Kalyan S Perumalla, and Sudip K Seal. Efficient simulation of agent-based models on multi-gpu and multi-core clusters. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 29. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010. 35, 59, 79, 102

[5] A.F. Abdelrazek, M. Kaschub, C. Blankenhorn, and M.C. 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. 36, 103

[6] Jeff Ahrenholz, Claudiu Danilov, Thomas R Henderson, and Jae H Kim. Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7. IEEE, 2008. 13

[7] P. Andelfinger, J. Mittag, and H. Hartenstein. Gpu-based architectures and their benefit for accurate and efficient wireless network simulations. In *MASCOTS, 2011 IEEE 19th International Symposium on*, pages 421–424. IEEE, 2011. 32

[8] J. April, F. Glover, J.P. 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. 36, 102

[9] Manish Arora. The architecture and evolution of cpu-gpu systems for general purpose computing. *By University of California, San Diago*. 27

[10] S. Bai and D.M. Nicol. Acceleration of wireless channel simulation using gpus. In *Wireless Conference (EW), 2010 European*, pages 841–848. IEEE, 2010. 32, 36, 103

[11] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In vini veritas: realistic and controlled network experimentation. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 3–14. ACM, 2006. 12

[12] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems*, pages 359–371, 2011. 67

[13] Bilel Ben Romdhanne, Mohamed Said Mosli Bouksiaa, Navid Nikaein, and Christian Bonnet. Hybrid scheduling for event-driven simulation over heterogeneous computers. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pages 47–56. ACM, 2013. 85

[14] Claude Berrou and Alain Glavieux. Near optimum error correcting coding and decoding: Turbo-codes. *Communications, IEEE Transactions on*, 44(10):1261–1271, 1996. 21

[15] B.R. 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. 35, 46, 102

[16] B.R. Bilel, N. Navid, and M.S.M. Bouksiaa. Hybrid cpu-gpu distributed framework for large scale mobile networks simulation. In *Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on*, pages 44–53. IEEE, 2012. 67, 68, 93

[17] B.R. Bilel, N. Navid, and Bonnet C. Coordinator-master-worker model for efficient large scale network simulation. In *6th International ICST Conference on Simulation Tools and Techniques*, 2013. 67, 80, 93, 95, 103

[18] B.R. Bilel, N. Navid, K. R., and B. C. Openairinterface large-scale wireless emulation platform and methodology. In *MSWIM*. ACM, 2011. 13, 36, 44, 103, 125

[19] K.C. Borries, G. Judd, D.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. 13, 36, 102

[20] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Micro, IEEE*, 32(2):28–37, 2012. 27

[21] T.D. Braun, H.J. Siegel, N. Beck, L.L. Bölöni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001. 59

[22] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, et al. Advances in network simulation. *Computer*, 33(5):59–67, 2000. 13

[23] H. Breu and D.G. Kirkpatrick. Unit disk graph recognition is np-hard. *Computational Geometry*, 9(1):3–24, 1998. 45

[24] Mark Carson and Darrin Santay. Nist net: a linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003. 13

[25] Franðcois E Cellier and Ernesto Kofman. *Continuous system simulation*. Springer, 2006. 14

[26] 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. 35, 80, 102

[27] Li-li Chen, Ya-shuai Lu, Yi-ping Yao, Shao-liang Peng, et al. A well-balanced time warp system on multi-core environments. In *Principles of Advanced and Distributed Simulation (PADS), 2011 IEEE Workshop on*, pages 1–9. IEEE, 2011. 34, 74, 75

[28] Matthew Chidester and Alan George. Parallel simulation of chip-multiprocessor architectures. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(3):176–200, 2002. 83

[29] NM Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010. 13

[30] E.S. Chung, E. Nurvitadhi, J.C. Hoe, B. Falsafi, and K. Mai. Protoflex: Fpga-accelerated hybrid functional simulator. In *IPDPS 2007. IEEE International*, pages 1–6. IEEE, 2007. 32

[31] Teodor Gabriel Crainic and Michel Toulouse. *Parallel strategies for meta-heuristics*. Springer, 2003. 56

[32] Tim Cramer, Dirk Schmidl, Michael Klemm, and Dieter an Mey. Openmp programming on intel r xeon phi tm coprocessors: An early performance comparison. 2012. 36, 103

[33] R. Curry, C. Kiddle, R. Simmonds, and B. Unger. Sequential performance of asynchronous conservative pdes algorithms. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 217–226. IEEE Computer Society, 2005. 33, 74

[34] Teresa A Dahlberg, Asis Nasipuri, and Craig Taylor. Explorebots: a mobile network experimentation testbed. In *Proceedings of the 2005 ACM SIGCOMM workshop on Experimental approaches to wireless network design and analysis*, pages 76–81. ACM, 2005. 12

[35] Judith S Dahmann. High level architecture for simulation. In *Distributed Interactive Simulation and Real Time Applications, 1997., First International Workshop on*, pages 9–14. IEEE, 1997. 14

[36] Gabriele D'Angelo and Michele Bracuto. Distributed simulation of large-scale and detailed models. *International Journal of Simulation and Process Modelling*, 5(2):120–131, 2009. 34, 75

[37] Peter J. Dennin. Is computer science science? *Commun ACM*, 48(4):27 – 31, 2005. 123

[38] K. Dragicevic and D. Bauer. A survey of concurrent priority queue algorithms. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6. IEEE, 2008. 33, 74

[39] EA. Electronic arts, 2013. 94

[40] Serge Fdida, Timur Friedman, and Sophia MacKeith. Onelab: Developing future internet testbeds. In *Towards a Service-Based Internet*, pages 199–200. Springer, 2010. 13

[41] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, 1993. 75

[42] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990. 21

[43] Richard M Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 46–53. IEEE Computer Society, 1999. 82

[44] Richard M Fujimoto. Parallel and distributed simulation. In *Simulation Conference Proceedings, 1999 Winter*, volume 1, pages 122–131. IEEE, 1999. 21

[45] Richard M Fujimoto. Parallel simulation: parallel and distributed simulation systems. In *Proceedings of the 33nd conference on Winter simulation*, pages 147–157. IEEE Computer Society, 2001. 43

[46] R.M. Fujimoto. Lookahead in parallel discrete event simulation. Technical report, DTIC Document, 1988. 34, 35, 75, 101

[47] R.M. Fujimoto, K. Perumalla, A. Park, H. Wu, M.H. Ammar, and G.F. 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. 21, 35, 59, 79, 101

[48] R.M. Fujimoto, K. Perumalla, A. Park, H. Wu, M.H. Ammar, and G.F. Riley. Large-scale network simulation: how big? how fast? In *11th IEEE/ACM MASCOTS 2003.*, pages 116 – 123, oct. 2003. 41, 79

[49] Stephen Bo Furber. *ARM system-on-chip architecture*. pearson Education, 2000. 27

[50] Erek Göktürk. A stance on emulation and testbeds, and a survey of network emulators and testbeds. *Proceedings of ECMS*, 2007. 13

[51] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. Time jails: A hybrid approach to scalable network emulation. In *Principles of Advanced and Distributed Simulation, 2008. PADS'08. 22nd Workshop on*, pages 7–14. IEEE, 2008. 14

[52] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C Snoeren, Amin Vahdat, and Geoffrey M Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM, 2005. 13

[53] Shashi Guruprasad, Robert Ricci, and Jay Lepreau. Integrated network experimentation using simulation and emulation. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2005. Tridentcom 2005. First International Conference on*, pages 204–212. IEEE, 2005. 15

[54] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. Experimental methodologies for large-scale systems: a survey. *Parallel Processing Letters*, 19(03):399–418, 2009. 124

[55] Linley Gwennap. Sandy bridge spans generations. *Microprocessor Report*, 9(27):10–01, 2010. 27

[56] IS Hammoodi, BG Stewart, A Kocian, and SG McMeekin. A comprehensive performance study of opnet modeler for zigbee wireless sensor networks. In *Next Generation Mobile Applications, Services and Technologies, 2009. NGMAST'09. Third International Conference on*, pages 357–362. IEEE, 2009. 14

[57] J. Harri, F. Filali, and C. Bonnet. Mobility models for vehicular ad hoc networks: a survey and taxonomy. *Communications Surveys & Tutorials, IEEE*, 11(4):19–41, 2009. 45

[58] Alexander Heinecke, K Vaidyanathan, M Smelyanskiy, A Kobotov, R Dubtsov, G Henry, AG Shet, G Chrysos, and P Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel (r) xeon phi (tm) coprocessor. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2013)*, 2013. 36, 103

[59] Stephen Hemminger et al. Network emulation with netem. In *Linux Conf Au*, pages 18–23. Citeseer, 2005. 13

[60] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and JB Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 2008. 14

[61] Thomas R Henderson, Sumit Roy, Sally Floyd, and George F Riley. ns-3 project goals. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 13. ACM, 2006. 106

[62] M. Hybinette and R.M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(4):378–407, 2001. 33, 74

[63] Natalie Ivanic, Brian Rivera, and Brian Adamson. Mobile ad hoc network emulation environment. In *Military Communications Conference, 2009. MILCOM 2009. IEEE*, pages 1–6. IEEE, 2009. 13

[64] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013. 36, 103

[65] David B Johnson. Validation of wireless and mobile network models and simulation. In *DARPA/NIST network simulation validation workshop*, 1999. 13

[66] Glenn Judd and Peter Steenkiste. Using emulation to understand and improve wireless networks and applications. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 203–216. USENIX Association, 2005. 13

[67] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998. 81

[68] William Kasch, J Ward, and Julia Andrusenko. Wireless network modeling and simulation tools for designers and developers. *Communications magazine, IEEE*, 47(3):120–127, 2009. 13

[69] Charles Peri Ken Renard and Jerry Clarke. A performance and scalability evaluation of the ns-3 distributed scheduler. The Workshop on ns-3 (WNS3)), 2012. 31, 32, 127

[70] Stratos Keranidis, Dimitris Giatsios, Thanasis Korakis, Iordanis Koutsopoulos, Leandros Tassiulas, Thierry Rakotoarivelo, and Thierry Parmentelat. Experimentation in heterogeneous european testbeds through the onelab facility: The case of planetlab federation with the wireless nitos testbed. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 338–354. Springer, 2012. 13

[71] Wolfgang Kiess and Martin Mauve. A survey on real-world implementations of mobile ad-hoc networks. *Ad Hoc Networks*, 5(3):324–339, 2007. 12

[72] Israel Koffman and Vincentzio Roman. Broadband wireless access solutions based on ofdm access in ieee 802.16. *Communications Magazine, IEEE*, 40(4):96–103, 2002. 21

[73] Georg Kunz, Olaf Landsiedel, Stefan Gotz, Klaus Wehrle, James Gross, and Farshad Naghibi. Expanding the event horizon in parallelized network simulations. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 172–181. IEEE, 2010. 106

[74] Georg Kunz, Mirko Stoffers, James Gross, and Klaus Wehrle. Runtime efficient event scheduling in multi-threaded network simulation. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 359–366. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011. 106

[75] Mathieu Lacage. Experimentation with ns-3. *Trilogy Summer School*, 2009. 14

[76] Mathieu Lacage. *Experimentation tools for networking research*. PhD thesis, Ph. D. dissertation, Ecole doctorale Stic, Université de Nice Sophia Antipolis, 2010. 37, 103

[77] Mathieu Lacage, Martin Ferrari, Mads Hansen, Thierry Turletti, and Walid Dabbous. Nepi: using independent simulators, emulators, and testbeds for easy experimentation. *ACM SIGOPS Operating Systems Review*, 43(4):60–65, 2010. 15

[78] Mathieu Lacage and Thomas R Henderson. Yet another network simulator. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 12. ACM, 2006. 90

[79] Chang Kil Lee and David Strang. The international diffusion of public-sector downsizing: Network emulation and theory-driven learning. *International Organization*, 60(4):883, 2006. 13

[80] Lawrence M Leemis and Stephen Keith Park. *Discrete-event simulation: A first course*. Pearson Prentice Hall Upper Saddle River, NJ, 2006. 19

[81] Olga Lesnova and Eugene Kalishenko. Ns-3 performance analysis and development of effective load balancing algorithms. 106

[82] Benyuan Liu, Yang Guo, James F Kurose, Donald F Towsley, and Weibo Gong. Fluid simulation of large scale networks: Issues and tradeoffs. In *PDPTA*, volume 99, pages 2136–2142, 1999. 43

[83] J. Liu. *Parallel Discrete-Event Simulation*. Wiley Online Library, 2009. 35, 101

[84] Jason Liu and Rong Rong. Hierarchical composite synchronization. In *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pages 3–12. IEEE, 2012. 34, 75

[85] Q. Liu and G. Wainer. Multicore acceleration of discrete event system specification systems. *Simulation*, 88(7):801–831, 2012. 35, 102

[86] Qi Liu. *Algorithms for parallel simulation of large-scale DEVS and Cell-DEVS models*. PhD thesis, Citeseer, 2010. 21

[87] Henrik Lundgren, David Lundberg, Johan Nielsen, Erik Nordstrom, and Christian Tschudin. A large-scale testbed for reproducible ad hoc protocol evaluations. In *Wireless Communications and Networking Conference, 2002. WCNC2002. 2002 IEEE*, volume 1, pages 412–418. IEEE, 2002. 14

[88] Huiwei Lv, Yuan Cheng, Lu Bai, Mingyu Chen, Dongrui Fan, and Ninghui Sun. P-gas: Parallelizing a cycle-accurate event-driven many-core processor simulator using parallel discrete event simulation. In *Principles of Advanced and Distributed Simulation (PADS), 2010 IEEE Workshop on*, pages 1–8. IEEE, 2010. 34, 75

[89] Steffen Maier, Daniel Herrscher, and Kurt Rothermel. Experiences with node virtualization for scalable network emulation. *Computer Communications*, 30(5):943–956, 2007. 13

[90] Michele Migliore, C Cannia, William W Lytton, Henry Markram, and Michael L Hines. Parallel network simulations with neuron. *Journal of computational neuroscience*, 21(2):119–129, 2006. 83

[91] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008. 53

[92] David M Nicol. Principles of conservative parallel simulation. In *Proceedings of the 28th conference on Winter simulation*, pages 128–135. IEEE Computer Society, 1996. 87

[93] Brian D Noble, Mahadev Satyanarayanan, Giao T Nguyen, and Randy H Katz. Trace-based mobile network emulation. *ACM SIGCOMM Computer Communication Review*, 27(4):51–61, 1997. 13

[94] C. Nvidia. Compute unified device architecture programming guide. *NVIDIA: Santa Clara, CA*, 2011. 32

[95] A. Park and R.M. 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. 35, 102

[96] Alfred Park and Ric Fujimoto. Efficient master/worker parallel discrete event simulation. *2009 ACMIEEESCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 145–152, 2009. 32, 44

[97] Alfred Park and Richard M. Fujimoto. Parallel discrete event simulation on desktop grid computing infrastructures. *International Journal of Simulation and Process Modelling*, 5(2):157 – 171, 2009. 31

[98] Alfred J. Park and Richard M. Fujimoto. Efficient master/worker parallel discrete event simulation on metacomputing systems. *IEEE Transactions on Parallel and Distributed Systems*, 23:873–880, 2012. 79

[99] H. Park and P.A. Fishwick. A gpu-based application framework supporting fast discrete-event simulation. *Simulation*, 86(10):613–628, 2010. 34, 59, 75

[100] H. Park and P.A. Fishwick. An analysis of queuing network simulation using gpu-based hardware acceleration. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 21(3):18, 2011. 34, 69, 74, 75

[101] Hyungwook Park and Paul A. Fishwick. An analysis of queuing network simulation using gpu-based hardware acceleration. *ACM Trans. Model. Comput. Simul.*, 21(3), February 2011. 32

[102] J. Parker and J.M. Epstein. A distributed platform for global-scale agent-based models of disease transmission. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 22(1):2, 2011. 34, 75

[103] Sudeep Pasricha and Nikil Dutt. *On-chip communication architectures: system on chip interconnect*. Morgan Kaufmann, 2010. 27

[104] David Patterson. The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges. *NVIDIA Whitepaper*, 2009. 47

[105] J. Pelkey and G. Riley. Distributed simulation with mpi in ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 410–414. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011. 69

[106] SJ Pennycook, SD Hammond, SA Jarvis, and GR Mudalige. Performance analysis of a hybrid mpi/cuda implementation of the naslu benchmark. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):23–29, 2011. 35, 102

[107] K. Perumalla, R. Fujimoto, T. McLean, and G. Riley. Experiences applying parallel and interoperable network simulation techniques in on-line simulations of military networks. pages 97–104, 2002. 31

[108] Kalyan S Perumalla. Switching to high gear: Opportunities for grand-scale real-time parallel simulations. In *Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 3–10. IEEE Computer Society, 2009. 36, 59, 103

[109] K.S. Perumalla. Discrete-event execution alternatives on general purpose graphical processing units (gpgpus). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 74–81. IEEE Computer Society, 2006. 32

[110] K.S. Perumalla. Parallel and distributed simulation: traditional techniques and recent advances. In *Proceedings of the 38th conference on Winter simulation*, pages 84–95. Winter Simulation Conference, 2006. 21, 33, 35, 74, 101

[111] P. Peschlow, M. Geuer, and P. Martini. Logical process based sequential simulation cloning. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 237–244. IEEE, 2008. 33, 74

[112] Larry Peterson, Andy Bavier, Marc E Fiuczynski, and Steve Muir. Experiences building planetlab. In *Proceedings of the 7th symposium on*

Operating systems design and implementation, pages 351–366. USENIX Association, 2006. 13

[113] Larry Peterson and Timothy Roscoe. The design principles of planetlab. *ACM SIGOPS Operating Systems Review*, 40(1):11–16, 2006. 14

[114] Francesco Quaglia and Vittorio Cortellessa. Grain sensitive event scheduling in time warp parallel discrete event simulation. In *Proceedings of the fourteenth workshop on Parallel and distributed simulation*, pages 173–180. IEEE Computer Society, 2000. 82

[115] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. Omf: a control and management framework for networking testbeds. *ACM SIGOPS Operating Systems Review*, 43(4):54–59, 2010. 12

[116] Dipankar Raychaudhuri, Ivan Seskar, Max Ott, Sachin Ganu, Kishore Ramachandran, Haris Kremo, Robert Siracusa, Hang Liu, and Manpreet Singh. Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 3, pages 1664–1669. IEEE, 2005. 12

[117] Shafqat Rehman, Thierry Turletti, Walid Dabbous, et al. A roadmap for benchmarking in wireless networks. 2011. 4

[118] George F Riley. Simulation of large scale networks ii: large-scale network simulations with gtnets. In *Proceedings of the 35th conference on Winter simulation: driving innovation*, pages 676–684. Winter Simulation Conference, 2003. 31

[119] George F Riley, Mostafa H Ammar, Richard M Fujimoto, Alfred Park, Kalyan Perumalla, and Donghua Xu. A federated approach to distributed network simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 14(2):116–148, 2004. 81

[120] Stewart Robinson. *Simulation: the practice of model development and use*. Wiley. com, 2004. 19

[121] Bilel Ben Romdhanne, Diego Dujovne, Thierry Turletti, Walid Dabbous, et al. Efficient and scalable merging algorithms for wireless traces. 2009. 4

[122] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(2):157–209, 1997. 33, 34, 74

[123] N. Satish, C. Kim, J. Chhugani, A.D. Nguyen, V.W. Lee, D. Kim, and P. Dubey. Fast sort on cpus, gpus and intel mic architectures. Technical report, Technical report, Intel, 2010. 35, 102

[124] Erik Saule, Kamer Kaya, and Umit V Catalyurek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. *arXiv preprint arXiv:1302.1078*, 2013. 36, 103

[125] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002. 56

[126] Guillaume Seguin. Multi-core parallelism for ns-3 simulator. *INRIA Sophia-Antipolis, Tech. Rep*, 2009. 106, 110

[127] Jafer Shafagh. *Parallel Simulation Techniques for Large-scale Discrete-event Models*. PhD thesis, Carleton University, 2011. 21, 43

[128] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru M Parulkar. Can the production network be the testbed? In *OSDI*, volume 10, pages 1–14, 2010. 12

[129] Edi Shmueli and Dror G Feitelson. Backfilling with lookahead to optimize the packing of parallel jobs. *Journal of Parallel and Distributed Computing*, 65(9):1090–1107, 2005. 88

[130] John A Sokolowski and Catherine M Banks. *Principles of modeling and simulation: a multidisciplinary approach*. Wiley. com, 2011. 14

[131] Tapas K Som and Robert G Sargent. A probabilistic event scheduling policy for optimistic parallel discrete event simulation. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 56–63. IEEE, 1998. 82

[132] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010. 26, 27

[133] Cristian Tala, Luciano Ahumada, Diego Dujovne, Shafqat-Ur Rehman, Thierry Turletti, and Walid Dabbous. Guidelines for the accurate design of empirical studies in wireless networks. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 208–222. Springer, 2012. 4

[134] W.T. Tang, R.S.M. Goh, and I.L.J. Thng. Ladder queue: An o (1) priority queue structure for large-scale discrete event simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(3):175–204, 2005. 33, 74

[135] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002. 59

[136] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002. 14

[137] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008. 14

[138] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pages 211–220. IEEE, 2012. 34, 75

[139] S.Y. Wang, C.C. Lin, Y.S. Tzeng, W.G. Huang, and T.W. Ho. Exploiting event-level parallelism for parallel network simulation on multicore systems. *Parallel and Distributed Systems, IEEE Transactions on*, 23(4):659–667, 2012. 34, 59, 75

[140] Philippe Wauteleta and Pierre Kestenera. Parallel io performance and scalability study on the prace curie supercomputer. *White paper, Prace*, 2011. 115

[141] Kirk Webb, Mike Hibler, Robert Ricci, Austin Clements, and Jay Lepreau. Implementing the emulab-planetlab portal: Experience and lessons learned. In *Proc. WORLDS*, 2004. 127

[142] Klaus Wehrle, Mesut Gèuneðs, and James Gross. *Modeling and tools for network simulation*. Springer, 2010. 19

[143] E. Weingartner, H. Vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. Ieee, 2009. 48, 50, 106

[144] T. Wenjie, Y. Yiping, and Z. Feng. A hierarchical parallel discrete event simulation kernel for multicore platform. *Cluster Computing*, pages 1–9, 2012. 35, 82, 102

[145] Matthew Wolf, Zhongtang Cai, Weiyun Huang, and Karsten Schwan. Smartpointers: personalized scientific data portals in your hand. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 20–20. IEEE, 2002. 90

[146] M. Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010. 67

[147] E. Wynters. Parallel processing on nvidia graphics processing units using cuda. *Journal of Computing Sciences in Colleges*, 26(3):58–66, 2011. 26

[148] Joshua J Yi and David J Lilja. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. *Computers, IEEE Transactions on*, 55(3):268–280, 2006. 14

[149] Srikanth B Yoginath, Kalyan S Perumalla, and Brian J Henz. Runtime performance and virtual network control alternatives in vm-based high-fidelity network simulations. In *Proceedings of the Winter Simulation Conference*, page 247. Winter Simulation, 2012. 15, 37, 103

[150] Marcelo Yuffe, Ernest Knoll, Moty Mehalel, Joseph Shor, and Tsvika Kurts. A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264–266. IEEE, 2011. 27

[151] Xu Zhang, Bowei Wang, and Chenyao Geng. Gpu-based background generation method. In *Wireless Mobile and Computing (CCWMC 2011), IET International Communication Conference on*, pages 117–120. IET, 2011. 27