MATHEUS LEÔNIDAS SILVA

Advisor: Joubert de Castro Lima

SIMPLICITY, REPRODUCIBILITY AND SCALABILITY FOR HUGE WIRELESS SENSOR NETWORK SIMULATIONS

Ouro Preto

May of 2018

Federal University of Ouro Preto
Institute of Exact Sciences
Master's degree in Computer Science

SIMPLICITY, REPRODUCIBILITY AND SCALABILITY FOR HUGE WIRELESS SENSOR NETWORK SIMULATIONS

Master's thesis presented to the Post-Graduation Program in Computer Science of the Federal University of Ouro Preto, as a partial requirement to obtain a Master's degree in Computer Science.

MATHEUS LEÔNIDAS SILVA

Ouro Preto
May of 2018

S381s Silva, Matheus Leônidas.

Simplicity, reproducibility and scalability for huge wireless sensor network simulations [manuscrito] / Matheus Leônidas Silva. - 2018.

61f.: il.: color; grafs; tabs.

Orientador: Prof. Dr. Joubert de Castro Lima.

Dissertação (Mestrado) - Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação. Programa de Pós-Graduação em Ciência da Computação.

Área de Concentração: Ciência da Computação.

1. Simulação (Computadores). 2. Computação de alto desempenho . 3. Sensoriamento remoto. I. Lima, Joubert de Castro. II. Universidade Federal de Ouro Preto. III. Titulo.

CDU: 004:528.835

Catalogação: www.sisbin.ufop.br

Ata da Defesa Pública de Dissertação de Mestrado

Aos 20 dias do mês de junho de 2018, às 14:00 horas na Sala de Seminários do DECOM no Instituto de Ciências Exatas e Biológicas (ICEB), reuniram-se os membros da banca examinadora composta pelos professores: **Prof. Dr. Joubert de Castro Lima (presidente e orientador), Prof. Dr. Antonio Alfredo Ferreira Loureiro e Prof. Dr. André Luiz Lins de Aquino**, aprovada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação, a fim de arguirem o mestrando **Matheus Leônidas Silva**, com o título **"Simplicity, Reproducibility and Scalability for Huge Wireless Sensor Network Simulations"**. Aberta a sessão pelo presidente, coube ao candidato, na forma regimental, expor o tema de sua dissertação, dentro do tempo regulamentar, sendo em seguida questionado pelos membros da banca examinadora, tendo dado as explicações que foram necessárias.

Recomendações da Banca:

[✓ (X) Aprovada sem recomendações]

( ) Reprovada

( ) Aprovada com recomendações: ______________________________

Banca Examinadora:

[illegible]
Prof. Dr. Joubert de Castro Lima

****
Prof. Dr. Antonio Alfredo Ferreira Loureiro

****
Prof. Dr. André Luiz Lins de Aquino

Certifico que a defesa realizou-se com a participação à distância dos membros Prof. Dr. Antonio Alfredo Ferreira Loureiro e Prof. Dr. André Luiz Lins de Aquino, depois das arguições e deliberações realizadas, os participantes à distância estão de acordo com as recomendações da banca examinadora.

[illegible]
Prof. Dr. Joubert de Castro Lima

[illegible]
Prof. Dr. Joubert de Castro Lima
Coordenador do Programa de Pós-Graduação em Ciência da Computação
DECOM/ICEB/UFOP
Ouro Preto, 20 de junho de 2018.

Resumo

Neste trabalho apresentamos duas contribuições para a literatura de redes de sensores sem fio (WSN). A primeira é um modelo geral para alcançar a reprodutibilidade no nível do kernel em simuladores paralelos. Infelizmente, os usuários devem implementar do zero como suas simulações se repetem em simuladores WSN, mas uma simulação paralela ou distribuída impõe o princípio de concorrência, não trivial de ser implementada por não especialistas. Testes usando o simulador chamado *JSensor* comprovaram que o modelo garante o nível mais restrito de reprodutibilidade, mesmo quando as simulações adotam diferentes números de threads ou diferentes máquinas em múltiplas execuções. A segunda contribuição é o simulador *JSensor*, um simulador paralelo de uso geral para aplicações WSN de grande escala e algoritmos distribuídos de alto nível. O *JSensor* introduz elementos de simulação mais realistas, como o ambiente representado por células personalizáveis e eventos de aplicação que representam fenômenos naturais, como raios, vento, sol, chuva e muito mais. As células são colocadas em uma grade que representa o ambiente com características do espaço definido pelos usuários, como temperatura, pressão e qualidade do ar. Avaliações experimentais mostram que o *JSensor* tem boa escalabilidade em arquiteturas de computadores multi-core, alcançando um *speedup* de 7,45 em uma máquina com 16 núcleos com tecnologia *Hyper-Threading*, portanto 50% dos núcleos são virtuais. O JSensor também provou ser 21% mais rápido que o *OMNeT++* ao simular um modelo do tipo flooding.

Abstract

In this work we present two contributions for the wireless sensor network (WSN) literature. The first one is a general model to achieve reproducibility in kernel level of parallel simulators. Unfortunately, users must implement how their simulations repeat from scratch in WSN simulators, but a parallel or distributed simulation imposes the concurrence principle, not trivial to be implemented by non-specialists. Tests using the simulator named JSensor proved that the model guarantees the most restrict level of reproducibility, even when simulations adopt different number of threads or different machines in multiple runs. The second contribution is the JSensor simulator, a parallel general purpose simulator for large scale WSN applications and high-level distributed algorithms. JSensor introduces more realistic simulation elements, such as the environment represented by customizable cells and application-events representing natural phenomena, such as lightning, wind, sun, rain and more. The cells are placed in a grid that represents the environment with characteristics of the space defined by the users, such as temperature, pressure and air quality. Experimental evaluations show that JSensor has good scalability in multi-core computer architectures, achieving a speedup of 7.45 in a machine with 16 cores with Hyper-Threading Technology, thus 50% of cores are virtual ones. JSensor also proved to be 21% faster than OMNeT++ while simulating a flooding model.

This thesis is dedicated to my father, who taught me that the best kind of knowledge to have is that which is learned for its own sake. It is also dedicated to my mother, who taught me that even the largest task can be accomplished if it is done one step at a time.

Acknowledgments

First and foremost, I have to thank my advisor, Dr. Joubert C. Lima. Without his assistance and dedicated involvement in every step throughout the process, this project would have never been accomplished. I would like to thank you very much for your support and understanding over these past years.

Getting through my thesis required more than academic support, and I have many, many people to thank for listening to and, at times, having to tolerate me over the past years. I cannot begin to express my gratitude and appreciation for their friendship.

Most importantly, none of this could have happened without my family. To my parents and my sister– it would be an understatement to say that, as a family, we have experienced some ups and downs in the past years. Every time I was ready to quit, you did not let me and I am forever grateful. This dissertation stands as a testament to your unconditional love and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

How can we analyze the water quality of a river? How can we monitor streets, cities, provinces of a country? How can we predict and prevent problems coming from earthquakes, snowstorms, fire spreads and all kinds of natural disasters? How can we test some new drugs without using animals? These and many other problems are initially solved by using simulators. It is extremely expensive, and sometimes dangerous, to test initial ideas in a real environment. Thus, very often we must use simulators to achieve comfortable confidence levels before real tests.

The use of simulators that until few years ago was concentrated in military and scientific domains have recently become important tools for many diverse areas, including environmental monitoring, medical (drugs utilization, clinical feedbacks, etc.), human occupations in risk, professional skills training, sports and so forth Okuda et al. (2009); Raković and Lutovac (2015); Gregory (2017). One way to simulate abstractions and their interactions is modeling them as sensor networks or precisely wireless sensor networks since some abstractions are mobile. These abstractions are named nodes and they are capable of sensing, actuating, processing, communicating and moving around the environment. Such nodes or sensors or devices, working cooperatively, are termed wireless sensor networks (WSNs) Akyildiz et al. (2002a). WSN simulators Varga and Hornig (2008a); Xue et al. (2007); Zeng et al. (1998a); Ould-Ahmed-Vall et al. (2007a) designed for huge applications Chen et al. (2005), where thousands or even millions of nodes are simulated, are becoming regular with the advance of new WSNs applications, e.g., general Internet of Things (IoT) applications Muruganandam et al. (2018), smart cities Lu et al. (2017), social sensing Ali et al. (2011) or natural environment monitoring Lin et al. (2015).

WSNs simulators for huge applications are becoming essential, as mentioned before; however, standards and market-leaders simulators, such as OMNeT++, NS, GlomoSim and others Varga and Hornig (2008b); Xue et al. (2007); Zeng et al. (1998a); Ould-Ahmed-Vall et al. (2007a), do not implement environment modeling and application-events to simulate interferences from external agents or natural phenomena, such as rain, sun, lightning, and so forth, in Earth regions. The work Sundresh et al. (2004) detailed the importance of environment modeling in WSN simulations, despite of most simulators, in special high performance computing (HPC) ones, do not consider such a concept. Furthermore, some HPC simulators, like the PDNS Riley and Park (2004a), exposes its distribution complexities, such as the concept of messages of MPI Pacheco (1997) libraries, to their users, so they require high performance programming skills, a non-trivial programming knowledge for non computer scientists like the simulator users.

Another essential feature is to repeat a simulation, but this obvious requirement was not addressed by the WSN simulator literature until now. In general, the users must implement how their simulations will repeat from scratch. This development strategy implies in serious drawbacks since the users normally put together business domain issues with reproducibility issues in the simulation code. Another limitation occurred while using a parallel or distributed version of a simulator because the users will require high performance programming skills to implement their reproducibility demands. These skills include understanding concurrence principle, its benefits and limitations, a deep technical knowledge even for computer scientists.

To attenuate the reproducibility problem in WSN simulators, this work presents the first generic reproducibility model to be used in kernel level of a parallel WSN simulator, this way not requiring any user interference to guarantee repeatable simulation runs, even when we change the hardware or the simulator configuration after each run (Ex. use a quadcore machine in run one and an octocore in run 2 or set the simulator with four threads in run one and with sixteen threads in run two). The model implements the concept of chunk to allow reproducibility, even when executing non-deterministic simulations in parallel. A chunk encapsulates a set of nodes and application-events, a seed to ensure a starting point to generate pseudo-random numbers and a generator of pseudo-random numbers. The reproducibility model has the fundamental ideas to partition the data and the processing of a simulation among many threads, trying to achieve a fair load balancing and consequently scaling up the simulations. To evaluate its applicability and correctness, we have implemented the reproducibility model in Java and integrated it with *JSensor* simulator Silva et al. (2013); Ribeiro et al. (2012b,a). This research result is detailed in Chapter 2.

The second contribution of this work is a new WSN simulator to attenuate the limitations mentioned before. The new simulator is an extension of *JSensor* simulator Silva et al. (2013); Ribeiro et al. (2012b,a), thus it is a Java parallel simulator to multi-core HPC architectures that handles millions of elements, performing the evaluation of low level protocols, WSN applications and high-level distributed algorithms according to the user needs. *JSensor* types and operators are designed to be both simple and extensible to users, hiding HPC complexities, but also capable to scale over multi-core machines. It implements environment modeling and application-events to simulate interferences from external agents or natural phenomena.

Experimental evaluations were done to discover the simulator scalability, its memory consumption and if it is faster than a literature counterpart, precisely the OMNET++ simulator. The performance results achieved speedups of 7.45 with 16 threads running concurrently and the comparative evaluations against OMNeT++ reveled a runtime 21% faster of *JSensor* when simulating the flooding model where the messages spreads to all connected nodes. This technological result is detailed in Chapter 3.

After explaining both contributions, their benefits and limitations, we conclude the work, presenting the final discussions and future investigations in Chapter 4.

Chapter 2

Reproducibility in Parallel Sensor Network Simulations

2.1 Abstract

Currently, some network simulators adopt parallel computer architectures to improve their scalability. The main problem with this strategy is guaranteeing the reproducibility transparently to simulation users. To diminish this problem, in this work, we present a reproducible parallel simulation model that can be adopted in huge sensor network scenarios. This model was integrated into a parallel sensor network simulator and validated accordingly. The model is based on a chunk partition strategy, in which all simulation elements are wrapped into chunks and simulated sequentially inside each chunk. Multiple chunks can be simulated in parallel as many times as necessary by adding a seed and a pseudo-random number generator in each of them, thus always ensuring the same results. The results demonstrated that our model could guarantee the reproducibility of stochastic parallel simulations performed in different computer systems with different numbers of threads.

2.2 Introduction

A fundamental simulation requirement is reproducibility. Among the different reproducibility levels Dalle (2012a), the most restricted is repeatability, which allows the re-execution of exactly the same simulation in terms of computation. In a discrete-event simulation, repeatability means that the simulations produce the same series of events and processes in the same order. In the case of parallel simulations, repeatability implies that concurrent activities must be always processed in the same order, which requires additional synchronization and sorting techniques. In general, parallel simulators do not support reproducibility in their kernels, leaving this activity to their users.

In this work, we present a reproducible parallel-simulation model to be used on sensor network scenarios that demand thousands of devices (Rashid and Rehmani, 2016; Aquino and Nakamura, 2009; Maia et al., 2013). The nature of these networks, with thousands of sensors/devices operating concurrently, and the reproducibility requirements are the biggest motivations for the proposed study. Basically, our model consists of wrapping all simulation abstractions into several chunks. These chunks are processed as ordinary sequential simulator kernels; i.e., all computations and communications of a simulation are performed from a single seed value and a pseudo-aleatory number generator. With this, it is possible to run chunks in parallel and guarantee that each chunk can repeat its execution because ensuring the reproducibility of sequential simulations is a trivial task.

Our model was integrated into the *JSensor* simulator Ribeiro et al. (2012c), which can run parallel simulations over shared-memory computer architectures. Thus, *JSensor* allows faster simulations and/or scenarios in which hundreds of thousands of sensors are simulated by a multi-core CPU. The proposed model guarantees *JSensor* reproducibility, including when it is executed with different numbers of threads. The reproducibility level achieved with the proposed model is repeatability. In this way, stochastic parallel simulations are feasible in *JSensor*. The results demonstrated that our model can guarantee reproducibility over the scenarios evaluated.

This chapter is organized as follows: Section 2.3 presents the related work. Section 2.4 formalizes the reproducibility model. Section 2.5 discusses the evaluations and results. Section 2.6 concludes the chapter and mentions some future work.

2.3 Related Work

We found several surveys describing different simulators used to evaluate sensor network applications Chhimwal et al. (2013a); Sundani et al. (2011a). Basically, a sensor network application is composed of i. manager-user interfaces; ii. high-level distributed algorithms; and iii. low-level protocols. In general, the simulators are used to evaluate high-level distributed algorithms or low-level protocols. In both cases, it is possible to simulate a complete application, but there are high development complexity costs.

To the best of our knowledge, *Java in Simulation Time* (JiST) Barr et al. (2005a) and *JSensor* Ribeiro et al. (2012c) are the only parallel simulators that specialize in high-level distributed algorithms. *JSensor* allows the development of simulation models considering some embedded software components, such as general data structures, inter-process communication mechanisms, and load balancing. As mentioned before, our model was integrated into this simulator and validated accordingly.

On the other hand, there are several parallel simulators that specialize in low-level protocols. Huge simulations have high processing costs, for instance, to simulate different network protocols simultaneously. Low-level parallel simulators include *Global Mobile System Simulator* (GloMoSim) Zeng et al. (1998b), *Parallel Distributed NS* (PDNS) Riley and Park (2004a), and *Georgia Tech Sensor Network Simulator* (GTSNetS) Ould-Ahmed-Vall et al. (2007b).

None of the well-established and market-leading simulators mention their reproducibility support on the kernel level, i.e., are transparent to the users. Normally, reproducibility issues are coded from scratch by the simulator users.

Considering reproducibility strategies on the user level, there are studies of methods to generate parallel random numbers Hill (2015); to improve numerical consistency for parallel calculation across a vast number of processors Robey et al. (2011a); and to achieve numerical reproducibility in Monte Carlo simulations Cleveland et al. (2013a). Based on the above discussion, we present a reproducibility model that can be adopted on the kernel level of a parallel simulator, hiding this development task.

2.4 Reproducibility model

Let the simulated elements be represented by: a set of sensors (S); a set of environmental models (E) to describe the time-space domain and characteristics of the monitored area; and a set of interaction models (Ψ) to describe the sensor-to-sensor and sensor-to-environment interactions. These simulated elements are defined and implemented by users and executed following a global time clock.

The proposed model is composed of three main artifacts: global, local and synchronization. The first describes the global simulation processing and it is formally represented as a 4-tuple, $(\mathcal{C}, Sy, \sigma, G)$, which consists of a set of chunks $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_c\}$ to wrap one or more instances of simulated elements (S, E and Ψ), where c is defined by the user; a synchronization element Sy to schedule the interactions among chunks, i.e., the Ψ instance events; a fixed global seed σ; and a global random-number generator $G(\sigma) = \{\sigma_{sy}, \sigma_1, \sigma_2, \ldots, \sigma_c\}$, where σ_{sy} is adopted by the synchronization element and the others ($\sigma_{1\ldots c}$) are considered in each chunk.

The operation of the global artifact is described as follows: i. t threads are created, $\mathbf{T} = \{T_0, T_1, T_2, \ldots, T_t\}$, where $t \leq c$ is defined by user. T_0 is the main simulation thread and it executes the synchronization element (Sy). $T_{1\ldots t}$ are the kernel threads and they encapsulate the chunks ($\mathcal{C}_{1\ldots c}$). ii. T_0 creates c chunks and allocates them for kernel threads ($T_{1\ldots t}$), following a circular list strategy. iii. At each simulation time (round), different simulated elements are instanced. These instances can be created statically or randomly, according to the user needs. In the static alternative, T_0 associates each new instance that is created before the simulation begins to the chunks sequentially, following a circular list strategy. New instances created during the simulation are allocated to the next chunk in the previous circular list. As all events are created in the same sequence, all chunks chosen will be the same in all simulations. The simulation execution time is defined by the user. In the random alternative, after the association following the same previous strategy, a simulated element of chunk $\mathcal{C}_i$ is instanced for execution in a random simulation time using seed σ_i. The random strategy guarantees the same association sequence when running different simulations. iv. the chunk interactions are coordinated by the synchronization element (Sy).

The local artifact describes the local simulation processing. This artifact is represented by the chunks. Each chunk ($\mathcal{C}_i$) is formally described as a 6-tuple, $(E', \Psi', S', \sigma_i, G)$, consisting of: a subset of the environmental models ($E' \subseteq E$); a subset of the sensor-interaction models ($\Psi' \subseteq \Psi$); a subset of the sensor nodes ($S' \subseteq S$); a fixed local seed (σ_i); and a local random number generator ($G(\sigma_i) = \{x_1, x_2, \ldots, x_l\}$). We can have one or more instances of simulated elements in a chunk; for instance, we can have only instances of sensor node elements or different instances of the environment, interactions and sensors. The simulated elements of different chunks are disjoint and must be the same in all simulations, regardless of the number of kernel threads.

The local artifact operation is determined by a synchronization element (Sy). When one instance of a simulated element is scheduled and released, it is executed sequentially. A simulated element of a chunk $\mathcal{C}_i$ generates interaction events based on seed σ_i, as described below. The interactions among the instances of simulated elements follow different random event times generated by $G(\sigma_{sy})$, as described below. These strategies guarantee the same creation and execution order when running different simulations.

Finally, the synchronization artifact determines the execution order and the synchronization of simulated element instances. This artifact is necessary because these instances must be able to both share and consume their information, which usually requires synchronization. The synchronization element (Sy) is formally described as a 3-tuple $(\mathbf{e}, \sigma_{sy}, G)$, consisting of: a set of events ($\mathbf{e} = \{\mathbf{e}_1 \cup \mathbf{e}_2 \cup \ldots \cup \mathbf{e}_k\}$), where k is the number of interaction instances (Ψ_k). Each subset $\mathbf{e}_k = \{e_{k,1}, e_{k,2}, \ldots, e_{k,m}\}$, where m is the number of events, describing the execution sequence of interaction instances; a fixed synchronization seed (σ_{sy}); and a synchronized random number generator $G(\sigma_{sy}) = \{y_1, y_2, \ldots, y_n\}$.

Each event $e_{k,m} = (y_n, id_k, id_m, msg, \mathcal{C}_{orig}, \mathcal{C}_{dest})$ is composed of: the random time-scheduling value (y_n); the event-type value (id_k) based on the Ψ_k interaction model; the event-identification value (id_m), generated randomly and based on random seed (σ_{orig}) of $\mathcal{C}_{orig}$; the message (msg) adopted to describe the event action and implemented by the user; the origin chunk $\mathcal{C}_{orig}$, where $1 \leq orig \leq c$; and the destination chunk $\mathcal{C}_{dest}$, where $1 \leq dest \leq c$ and $orig \neq dest$.

Let `inbox` be the list of running events ($e_{k,m}$), indexed by event type (id_k), and for each event type we have its ordered list of events indexed by event identification (id_m). The operation of the synchronization artifact at each global time clock is thus as follows: i. Different events are written in `inbox`, following their type (id_k). ii. All events are sorted according to identification (id_m). iii. Events of different types are executed sequentially; i.e., all events with $k = 1$ are executed, then all events with $k = 2$, and so on. iv. Events of equal type in different threads are executed in parallel.

The event scheduling is based on the global time clock and the event type. Consequently, events will be the same in all executions. This scheduling, based on two rules, is performed because inconsistencies occur only when we have events of different types. In this case, the order is fundamental. For instance, in our model, the sensor node can perform concurrent communication, but it cannot communicate and sense in the same clock time. In this case, concurrent communication could be realized only once by the user because collision is an expected behavior in wireless communication. Additionally, these two scheduling rules are mandatory because non-determinism is introduced by the OS threads running over a computer architecture. The simulator kernel threads can guarantee the synchronization, but they inherit the non-deterministic behavior of OS threads because the former are managed by the latter. This scheduling does not avoid non-deterministic behavior, but it always guarantees the reproducibility by ordering the events.

2.5 Reproducibility model evaluation

The implementation of our model hides the reproducibility aspect from JSensor users. To ensure the reproducibility of simulations, the users need to set the numbers of threads and chunks and the initial σ value and then implement the simulation elements. In this implementation, the number of threads could be equal to or less than the number of machine cores. As mentioned before, the number of chunks is never less than the number of threads defined by the users.

All simulations were performed in a machine equipped with an Intel Core i7-4710HQ CPU with hyper-threading technology, with each core operating at 2.50 GHz and 16 GB of RAM DDR3 1333 Mhz. The OS was Ubuntu 14.04.1 LTS 64 bits kernel 3.13.0-39-generic, and all experiments fit in RAM memory. All algorithms were compiled in Java 64 bits (JAVA 8 - update 101).

To evaluate our model on JSensor, we used a scenario to stress the simulator in terms of events and messages. The simulation elements of our scenario were as follows: Set of sensors $S = S_{mobile} \cup S_{gateway}$, where $S_{mobile} = sm_1, sm_2, \ldots, sm_{10,000}$ represents the mobile sensors and $S_{gateway} = sg_1, sg_2, \ldots, sg_{100}$ represents the relays nodes; Set of environment models $E = E_s \cup E_r \cup E_l$, where E_s is the sun radiation, E_r is the incidence of rain, and E_l the presence of lightning. The environment area $(10,000 \times 10,000)$ is partitioned in a grid of cells $(1,000 \times 1,000)$ to delimit the incidence of the phenomena. The mobile sensors S_{mobile} are deployed at random locations and the gateways $S_{gateway}$ are located at the center of each cell. The natural phenomena occur randomly during the simulation. There is a set of interaction models $\Psi = \Psi_r \cup \Psi_c \cup \Psi_m \cup \Psi_s$, where Ψ_r is the routing communication model (sensor-to-sensor), based on the Dynamic Source Routing (DSR) algorithm Johnson et al. (2001). Ψ_c is the physical communication model (sensor-to-sensor), based on Unit Disk Graph (UDG) connectivity, which adopts a communication range of $1,000$. Ψ_m is the sensor mobility model (sensor-to-environment), based on a random walk Angelopoulos et al. (2010). Ψ_s is the sensing model (environment-to-sensor). Additionally, we set $\sigma = 1587632589$ in all simulations.

We executed the same simulation, varying the number of threads (1, 2, 4, 8) and quantifying the number of messages, mobility events, and node deactivations. To identify reproducibility problems, we executed our implementation considering the following: i. different numbers of chunks (Chunks) ii. the use of only one seed to generate the random numbers (Seed); and iii. the original sequence of events, ignoring the ordering step (Sort). These different configurations were considered because they are the most common problems in the development of parallel simulations.

Tables 2.1 - 2.3 illustrate the reproducibility evaluation. The results demonstrated that our model maintains the same quantifier values when the number of threads varies. However, in all other configurations, the final quantifier values change, indicating a failure to reproduce the same simulation.

Table 2.1: Reproducibility evaluation :: Number of messages

#threads	Our model	Chunks	Seed	Sort
8	9,768	10,079	8,529	9,133
4	9,768	10,241	9,054	9,408
2	9,768	10,241	9,752	10,646
1	9,768	10,499	9,845	10,064

Table 2.1 presents the numbers of messages during the same simulated scenario, varying only the number of threads. We observed that when the simulated elements are created and allocated, specifically, into different numbers of chunks, the number of messages varies because the ordering of communication events is affected. For instance, the DSR algorithm generates more or fewer messages to discover an appropriate route because the order of messages arriving at a specific node could change in different simulations.

Table 2.2: Reproducibility evaluation :: Mobility events

#threads	Our model	Chunks	Seed	Sort
8	1,505,846	1,504,906	1,508,174	1,512,714
4	1,505,846	1,488,315	1,487,004	1,503,054
2	1,505,846	1,488,315	1,457,107	1,472,960
1	1,505,846	1,492,151	1,525,011	1,513,041

Table 2.2 presents the results about the number of mobility events. We observed that

when using only one seed to generate the random numbers, the number of mobility events varies. This occurs because the event id, which is randomly generated, is not the same in all simulations; consequently, the event order used during the scheduling is not the same.

Table 2.3: Reproducibility evaluation :: Node deactivation

#threads	Our model	Chunks	Seed	Sort
8	49	50	47	45
4	49	59	49	53
2	49	59	39	50
1	49	52	49	47

Finally, Table 2.3 illustrates the results on the number of node deactivations because this behavior is mandatory in several simulations. In our example, the environmental event lightning deactivates gateway sensors for a short period. We observed a difference in event ordering in different simulations when the synchronization mechanism is active but the sorting is not. These results demonstrated that sorting must work together with the synchronization strategy to achieve parallel reproducibility. The justification, as mentioned above, is the non-determinism of the OS threads. Specifically, the event order is not the same in different simulations; consequently, the event sequence in the environment cells does not follow the same order.

2.6 Conclusion

We presented a reproducible parallel simulation model to be adopted in sensor network scenarios. Our model consists of wrapping all simulation elements into several chunks. All computations and communications of a simulation are performed from a single seed value and a pseudo-aleatory number generator. Consequently, it is possible to run chunks in parallel and to guarantee that each chunk can repeat its execution.

We integrated our model into the *JSensor* simulator, which performs efficiently in multi-core architectures. By using our model, the reproducibility on JSensor was guaranteed. The results demonstrated that our model could guarantee the reproducibility when different parallel simulations are performed, including when different numbers of threads are used. Future works include applying this model in GPU-based simulations and other parallel simulations scenarios. Finally, our model could be integrated and evaluated in other parallel sensor network simulators.

Chapter 3

JSensor: A Parallel Simulator for Huge Wireless Sensor Network Simulations

3.1 Abstract

This work presents *JSensor*, a parallel general purpose simulator which enables huge simulations of wireless sensor networks (WSNs) applications. Its main advantages are: i) to have a simple API with few classes to be extended, allowing easy prototyping and validation of WSNs applications and protocols; ii) to enable transparent and reproducible simulations, regardless the number of threads of the parallel kernel; and iii) to scale over multi-core computer architectures, allowing more the simulation of more realistic elements, such as the environment and application-events representing natural phenomena. *JSensor* is a parallel event drive simulator which executes according to event timers, so the simulation elements (nodes, application or events) can send messages, process task or move around the environment. The mentioned environment follows a grid structure of extensible spatial cells. The results demonstrated that *JSensor* scales well, precisely it achieved a speedup of 7.45 with 16 threads in a machine with 16 cores (eight physical and eight virtual cores), and comparative evaluations versus OMNeT++ showed that the presented solution could be 21% faster.

3.2 Introduction

The world around us has a variety of phenomena described by variables such as temperature, pressure, and humidity, which can be monitored by devices capable of sensing, actuating, processing, communicating and moving around the environments. Such devices, working cooperatively, are termed wireless sensor networks (WSNs) Akyildiz et al. (2002a). WSNs simulators for large applications Chen et al. (2005), where thousands or even millions of nodes are simulated, are becoming regular with the advance of new WSNs applications, e.g., general Internet of things (IoT) applications Muruganandam et al. (2018), smart cities Lu et al. (2017), social sensing Ali et al. (2011) or natural environment monitoring Lin et al. (2015).

It is not simple to perform simulations of large WSNs applications, neither to achieve good speedup when we use parallel strategies for high-performance computing (HPC) architectures. The new WSNs applications have demanded robust simulators, e.g., simulators capable of evaluating scenarios with millions of nodes with protocols and specific models implemented. For instance, to simulate human behavior during one week in Tokyo downtown, based on cell phone information, would require millions of nodes with different communication protocols and interference models Xiong et al. (2016).

Standards and market-leaders simulators, such as OMNeT++, NS, and others Varga and Hornig (2008b); Xue et al. (2007); Zeng et al. (1998a); Ould-Ahmed-Vall et al. (2007a), do not implement environment modeling and application-events to simulate interference from external agents or natural phenomena, such as rain, sun, lightning, and so forth, in Earth regions. Sundresh et al. Sundresh et al. (2004) detail the importance to model the complete application environment in a WSN simulation. They propose a model for indoor and outdoor environments made of tiles. Based on this simulator, it is possible to implement concrete, grass and walls tiles, each one with different signal interference and propagation. However, the most simulators, in special HPC ones, do not consider this modeling.

Another limitation in WSN simulator is the non-reproducibility. In general, the users diminishes the reproducibility problem while implementing their simulation models in the simulators. Parallel simulations impose an extra challenge regarding reproducibility because they introduce the concurrency principle when implementing threads or processes sharing common resources, thus generating non-deterministic behaviors. In this case, repeatable parallel simulations must treat the non-determinism. In summary, reproducibility can become very complicated in parallel scenarios, being not suitable to be designed by users.

To allow parallel simulations of massive, robust and reproducible WSN applications, we present *JSensor*. It is a Java parallel simulator to multi-core HPC architectures that handles millions of elements, evaluating both low-level protocols and WSN applications according to the user requirements. *JSensor* types and operators are designed to be extensible and straightforward to users, but also capable of scaling over multi-core machines. It presents several application models, precisely node mobility model, node connectivity model, node distribution model, message transmission model, interference model and reliability model. Therefore we can emulate even message lost and channel noise. All these models are transparently reproducible, regardless the hardware used or the number of threads used in each parallel simulation, so users do not need to implement any extra code to achieve reproducibility in *JSensor*.

To evaluate *JSensor*, we tested it with three different applications: i) a robust data propagation application using the flooding model; ii) a mobile phone application with communication interference and WSN routing protocols; and iii) a sophisticated air quality application regarding CO_2. We execute the experiments in two machines, one with 16 cores and other with 24 cores, both with hyper-threading technology (50% of all cores are virtual). The performance results achieved speedups of 7.45 with 16 threads running concurrently. The comparative evaluations of *JSensor* versus OMNeT++ reveled a runtime 21% faster of *JSensor* when simulating the flooding model.

The remainder of this chapter is as follows: Section 3.3 discusses *JSensor* related work, pointing out their benefits and limitations. Section 3.4 presents the *JSensor* kernel and its specificities. Section 3.5 explains the parallelism implementation adopted by *JSensor*. Section 3.6 shows the *JSensor* strategies that allow the reproducibility of parallel and large WSNs applications. Section 3.7 describes the applications used to test the simulator speedup and memory consumption. Section 3.8 presents the performance evaluations conducted and the results obtained. Finally, Section 3.10 concludes the chapter and presents some the future research directions.

3.3 Related Work

We evaluate several parallel simulators used to assess WSNs applications. Several surveys described different WSNs simulators, their limitations, and their improvements Weingärtner et al. (2009); Khan et al. (2011); Chhimwal et al. (2013b); Sundani et al. (2011b); Yick et al. (2008); Akyildiz et al. (2002b); Vieira et al. (2003); Yu and Jain (2011); Minakov et al. (2016). In general, a WSN application presented the following elements: i) manager interfaces, e.g., data visualization or node range controllers; ii) high level distributed algorithms, e.g., clustering, localization, or data aggregation strategies; and iii) low-level protocols, e.g., routing, duty cycle, or address naming protocols. Normally, the simulators are used to evaluate high level distributed algorithms Barr et al. (2005b); Distributed Computing Group (2015) or low level protocols Zeng et al. (1998a); Doerel (2009); Riley and Park (2004b); Ould-Ahmed-Vall et al. (2007a). In both cases, it is possible to simulate a complete WSN application.

We observe the following features to evaluate the simulators: i) multicore or multicomputer architecture support; ii) parallel programming abstractions for users; iii) the catalog of available network protocols; iv) environment modeling, where we represent continuous territories of Earth as a grid of cells to place the simulation nodes and application-events, enabling transparent concurrent thread-safe accesses to Earth regions to change their properties, such as the water level, lightning occurrence, temperature, luminosity and so one. v) reproducibility of simulations at the kernel level, therefore transparent for users; and vi) application-events to model natural phenomena, such as thunder, rain, wind, lightning, and others. Note that, some application-events can occupy vast regions of Earth, like wind or sun, thus many cells of the environment, but a node always holds a single cell per simulation time; vii. The simulator project artifacts, e.g., programming and installation guides for users, discussion lists, if the solution is open source or not, discontinued or active, free or commercial and many other project issues.

To the best of our knowledge, *Java in Simulation Time* (JiST) Barr et al. (2005b) is the unique parallel simulator specialized in high level distributed algorithms. It is based on discrete events and could be used combined with *Scalable Wireless Ad hoc Network Simulator* (SWANS) Liu et al. (2001). In this simulator, each network element is an entity that can create events to be simulated. It performs the interactions among entities after synchronization barriers. Out of synchronization barriers, entities are independent, allowing the simulator do scale up regardless the application domain. JiST meets the "Multi-core support" and "Project artifacts" features.

On the other hand, we found several parallel simulators specialized in low-level protocols. One of them is the *Global Mobile System Simulator - GloMoSim* Zeng et al. (1998a), which is designed with libraries for WSN simulations. It is based on parallel discrete events and uses a C-like parallel simulation language (PARSEC) Bagrodia et al. (1998). There is a set of modules where each of them simulates a specific network protocol in the communication stack. There are a large number of protocols available in the *GloMoSim library.* To minimize the overhead of large-scale WSNs simulations each kernel thread is responsible for simulating one layer of the communication protocol stack. *GloMoSim* meets the "Multi-core support" and "Network protocols" features.

The *GloMoSim* commercial version is the *Qualnet* Doerel (2009). It was released in 2000 by Scalable Network Technology (SNT)[1]. Some additional features in the commercial version are a GUI based simulation, high fidelity commercial protocols and device models, comparative performance evaluation of alternative protocols at each layer, built-in measurement on each layer, modular layer stack design, and support for multiple parallel simulation strategies. *Qualnet* meets the "Multi-core support", "Network protocols", and "Project artifacts" features.

A widely used simulator for WSNs application is *Castalia*. *Castalia* is based on the OMNeT++ platform Varga and Hornig (2008b) and can be used to test distributed algorithms or protocols in realistic wireless channel and radio models. It also implements MAC and routing protocols. It is necessary to integrate MPI in OMNeT++ platform to perform parallel simulations. Therefore, it becomes difficult for users without HPC skills. Another drawback in using MPI over multicore computer architectures is that most of its implementations are for multicomputer clusters with private memory abstractions. Therefore, network communications are considered even in a shared memory multicore architecture what introduces a considerable overhead. *Castalia* meets the "Network protocols" and "Project artifacts" features.

[1]`http://web.scalable-networks.com/`

Other parallel simulator, based on Network Simulator version 2 (NS-2) Xue et al. (2007), is the *Parallel Distributed NS* (PDNS) Riley and Park (2004b). It uses a conservative, blocking based, an approach for synchronization with a possible sub-optimal speedup. It inherits all protocols, models, and applications of NS-2. However, all complexity and program difficulties are inherited too. Vodel et al. Vodel et al. (2008) mention that "the integration with such a complex framework, like NS-2, constrains PDNS to several conceptual limitations". PDNS meets the "Multi-core support", "Multi-computer support", and "Network protocols" features.

The NS-3 Riley and Henderson (2010) is also a version of the NS family, which is intended to replace the predecessor, NS-2. It has a discrete-event simulation engine written in C++ and a modular object-oriented architecture. The NS-3 provides a range of traditional protocols, such as TCP/IP, IPv6, and IEEE802.11. The simulator supports parallel and distributed simulation using MPI. Another novelty is the possibility of using Python in conjunction with C++ in some parts of the simulation. NS-3 meets the "Multi-core support", "Multi-computer support", "Network protocols" and "Project artifacts" features.

Finally, we found the *Georgia Tech Sensor Network Simulator* (GTSNetS) Ould-Ahmed-Vall et al. (2007a). It allows the development and evaluation of algorithms for large-scale WSNs. GTSNetS is complete, providing different energy models, battery models, network protocols, application protocols, and tracing options. Furthermore, the users could easily extend or replace the available models for a specific requirement. GTSNetS reports that could handle 200 thousand nodes in a simulation Khan et al. (2011). GTSNetS meets the "Network protocols" and "Project artifacts" features.

A summary of presented features among cited works and *JSensor* is presented in Table 3.1. As observed, *JSensor* meets most of the presented features.

Table 3.1: Summary of related work features

Features	*JSensor*	JisT	GlomoSim	Qualnet	Castalia	PDNS	NS-3	GTSNetS
Multicore support	✓	✓	✓	✓	–	✓	✓	–
Multicomputer support	–	–	–	–	–	✓	✓	–
Parallel programming abstraction	✓	–	–	–	–	–	–	–
Network protocols*	✓	–	✓	✓	✓	✓	✓	✓
Environment modeling	✓	–	–	–	–	–	–	–
Application-events	✓	–	–	–	–	–	–	–
Kernel level reproducibility	✓	–	–	–	–	–	–	–
Project artifacts	✓	✓	–	✓	✓	–	✓	✓

* *JSensor* considers only the routing layer

Additional features regarding both environment modeling and application-events in WSN applications considered only by *JSensor* are: i. *Application-events*: *JSensor* reinforce the existence of application-events and not only nodes in a WSN simulation. In conjunction with nodes and the simulated environment, application-events introduce dynamics in a simulation with the possibility of natural phenomena modeling, this way not only nodes-environment interactions exist, but also application-events-environment ones. Different from other simulators, in *JSensor* this feature is transparent to users; and ii. *Environment modeling*: Very often in a simulation, it is necessary to change the environment, so this simulation element must be extensible and dynamic. Different of presented simulators, *JSensor* implements a thread-safe environment to simulate vast territories of Earth. To discretize continuous Earth regions, *JSensor* adopts a grid of cells with extensible properties, very similar to a raster representation of Geographic Information Systems (GIS).

3.4 *JSensor* Parallel Simulator

JSensor is coded in Java, which means it is portable to many platforms with a JVM running. It is designed to execute asynchronous simulations based on events. Its architecture, illustrated in Figure 3.1, is organized into rings, according to its behaviors and visibility to users.

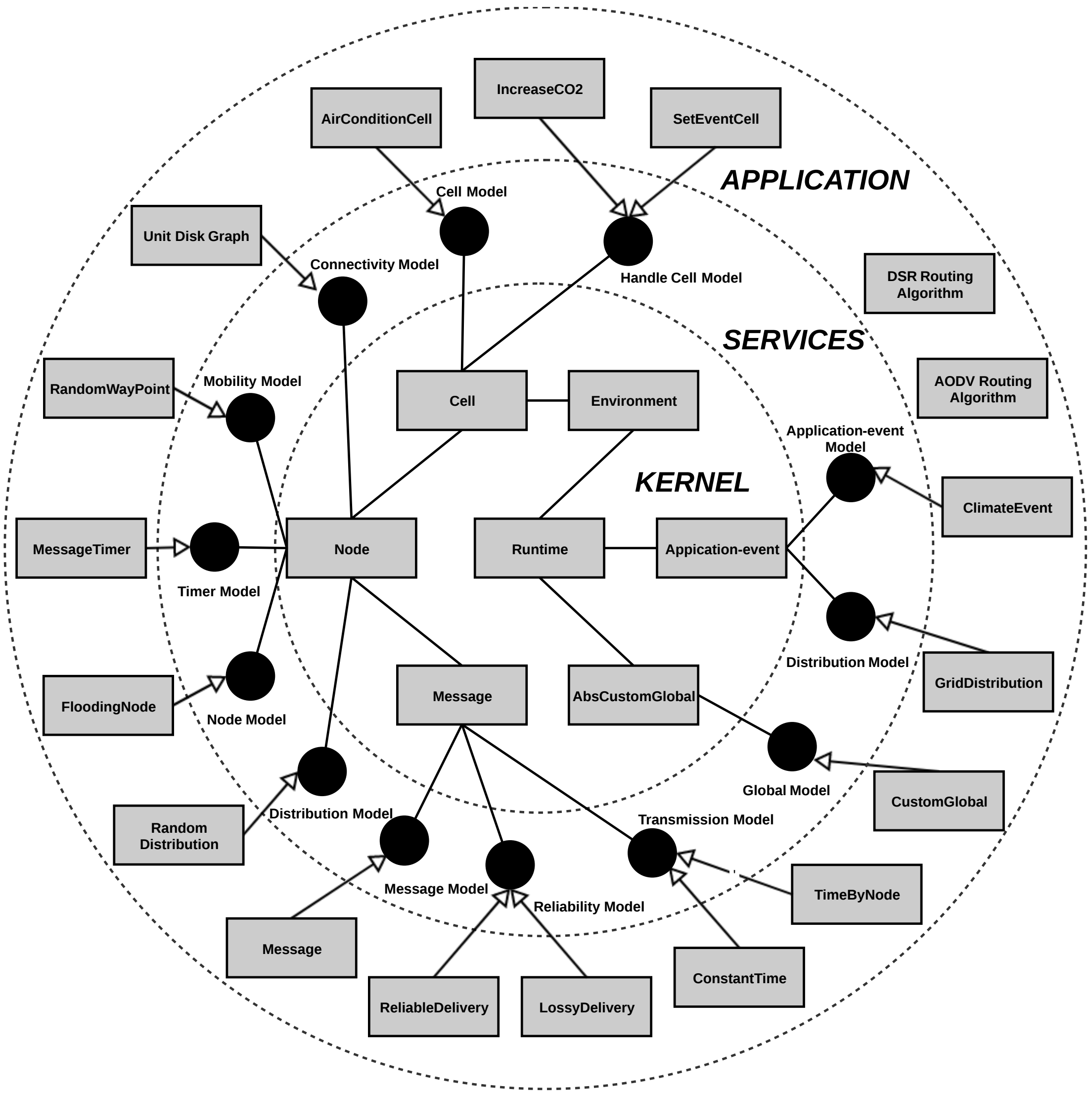

Figure 3.1: *JSensor*'s architecture

The more outer rings abstractions can adopt any inner ones so that the elements of the n-th circle can inherit behaviors from any inside ones. The most internal ring represents the kernel, where fundamental concepts are responsible for running simulations. The service layer provides different features for users develop their models and applications. Finally, the application layer represents the simulated application, each feature in *JSensor* has a practical example from an already developed model as illustrated in application ring on Figure 3.1.

3.4.1 Kernel Layer

The kernel layer creates the threads for parallel events execution, the chunks to guarantee reproducible parallel simulations with a different number of threads (detailed in Section 3.6), the environment which encapsulates a grid of cells, and the capability to log the simulation data. Furthermore, the kernel executes send-receive, timer, mobility and other simulation events of nodes and application-events. All simulation events have a predefined time to occur and *JSensor* guarantees their order by considering their internal clocks. Regularly, *JSensor* removes and executes events according to a global clock.

The kernel layer is composed by:

- `Runtime`: It is the main class, responsible for managing the structure of the kernel. It contains the lists of nodes and application-events, the chunks, and the environment. Furthermore, this class controls the threads that execute the simulation models, and it keeps the simulation times synchronized.

- `Node`: It represents the general node in a WSN application. Any new node type must extend the class `Node`. Nodes can have a variety of sensors, and they can move through the environment, interact with the environment, have an initial position, identify their neighbors, execute specific events, and communicate with other nodes. The communications are send-receive message events, classified in two types: *Unicast* and *Multicast*.

- `Message`: It is responsible for performing the communication between nodes. Using the *Unicast* or *Multicast* methods of the nodes, the message will be sent according to protocols, delays, package losses, channel noises, and any possible interference defined by the user. All new application messages must be a specialization of this class. Thus, it must implement a *clone()* method to enable copies of itself.

- `Application-event`: It represents an application phenomenon, natural or not, with lifetime, topology, mobility, and capacity of interacting with the environment. We observe the entire interleaving of events and nodes, e.g., when environmental changes occur, made by application-events, the nodes can perceive them.

- `Cell`: It represents a single element of the environment. It can hold different properties, like water level, humidity, CO_2 level and so forth. Nodes and application-events can read and write cell properties from regions where they live, concurrently and in a thread-safe manner. The Figure 3.2 illustrates that multiple nodes, application-events or their combinations are in a cell. They can read its properties and write new values into it, so the concurrent write operations are synchronized transparently by the kernel.

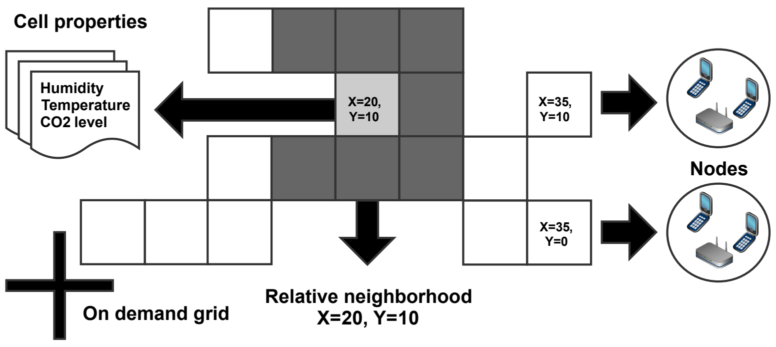

Figure 3.2: Grid of cells.

Despite the fact that the modification of a single cell is sequential, the cell modification step executes in parallel by splitting the cells into chunks, so all changes for a specific cell are made by the same chunk and, consequently, the same thread. The chunk concept and its benefits will be explained in details in Section 3.5.

- `Environment`: It is a map of spatial cells where the user can build his cells to customize the environment. A cell exists only if a node or an application-event have inhabited or modified it, in this way we build the *JSensor* environment on-demand. This process is essential for a simulator designed for large-scale scenarios, for instance, over vast territories of the Earth.

- `AbsCustomGlobal`: It maintains and controls general simulation timers. Besides, it enables the user to finalize the simulation and perform actions at key points, precisely before and after the execution of each simulation round and after the complete simulation.

3.4.2 Services Layer

JSensor provides a set of services for users so that they can develop their models for nodes, messages, cells, or application-events. To implement a new model, the users must extend *JSensor* existing types or implement different interface classes.

- `Node Model`: This interface is used to create a new Node. It contains the method *onCreation()*, which defines the first tasks during the node creation, and it is when the simulation starts. The second method, named *handleMessages(Inbox inbox)*, allows users to check the messages received in a specific event and to process them according to their needs.

- `Distribution Model`: The unique method of this interface is *getPosition(Node n)*, used to distribute the nodes through the environment during the initial state of the simulation, i.e., during the simulation first event. The method has a node as the argument and must return a valid position for this node.

- `Connectivity Model`: It represents the nodes interconnections. There are two types of connection between nodes, logical and physical. The logical one defines if a pair of nodes can communicate and its implementation is through the method *isConnected(Node from, Node to)*, which requires two nodes as arguments and returns a boolean value. The physical one occurs if two nodes are near enough to receive signals from each other and its implementation is through the method *isNear(Node n_1, Node n_2)*, requiring two nodes as arguments and returning a boolean value.

- `Message Model`: This interface allows users to create messages according to their needs. For that, it is necessary to implement the *clone()* method so that the kernel can create copies of the messages.

- `Timer Model`: Timers create new simulation events, allowing the simulation to schedule a task in the future.

- `Reliability Model`: It abstracts the reliability of the simulated network. For each message, the model decides if it should arrive at the destination or not. For several reasons a network packet cannot reach the destination, so *JSensor* enables such scenarios via reliability model implementations. The unique method necessary to build a new reliability model is the *reachesDestination(Message msg)* method, which returns a boolean value that indicates if the simulation should drop the message.

- `Transmission Model`: The message transmission model defines how much time the message spent until it reaches the destination. The method *timeToReach(Node nodeSource, Node nodeDestination, Message msg)* needs to be implemented and it returns a float value that represents the time.

- `Application-event Model`: It allows the user to create application-events, e.g., natural phenomena or any other external event capable of interacting with the environment. It requires the implementation of the method *setValue(CellModel cell)*, which argument is the cell to be modified. The method allows the user to make modifications in a cell returning the copy to be synchronized by the kernel in safe-mode. This strategy is mandatory to keep data consistency in parallel scenarios, where nodes and application-events change cells concurrently, even in non-commutative simulations, assuring the reproducibility.

- `Distribution Model`: By using this interface, the user can implement a distribution model for the application-events. The method *getPosition(Event e)* must be implemented and it returns a valid position for the application-event *e*.

- `Cell Model`: The users can create cell models according to their needs. The unique method to be implemented is the *clone()* method, which is adopted by the kernel to create copies of the cells to perform safe modifications.

- `Handle Cell Model`: The application of this model changes cells, keeping the consistency of the simulation state and also the reproducibility since the simulator must replace the cells in the same order, regardless it is an aleatory order or not. Due to *JSensor*'s parallel characteristic, the user must implement a handle model to keep the right order. The method to be executed is the *fire(Node node, CellModel cell)*, which has as arguments the node associated with a cell and the changed cell.

- `Mobility Model`: It defines how the nodes will move during the simulation, so to develop a new model the user must implement the methods *getNextPosition(Node n)* and *clone()*. The first one must return a valid position for the node *n* and the clone is adopted by the kernel to create copies of the model to perform parallel executions safely.

- `Global Model`: This model encapsulates several useful *JSensor* features. The method *hasTerminated()* stops the simulation when it returns true. Furthermore, it has the methods *preRun()* and *postRun()*, which are executed before and after the simulation, respectively. In the same context, the methods *preRound()* and *postRound()* can be executed before and after each round. A round represents a complete cycle of steps, e.g., move, send a message and discover neighbors. Each of these steps can have zero or more events to be simulated.

3.4.3 Application Layer

JSensor implements different applications to offer a significant number of features for new users. The following features are available in *JSensor* through the Flooding, Mobile Phone, and Air Quality applications detailed on Section 3.7.

- The `FloodingNode` of the flooding simulation decides during the *onCreation()* method call if the node will send an initial message, how it will move and also the creation of a structure that will control the received messages to avoid handling messages already received. When *handleMessages(Inbox inbox)* is executed, the node controls the messages, labeling them as received or sending them to its neighbors.

- The Flooding application implements the `RandomDistribution` model, which generates pseudo-random numbers to define the position of the nodes.

- The default connectivity implementation on *JSensor* is the `UnitDiskGraph` connectivity, which connects the nodes if they are inside each other's radius. The flooding application connects all the nodes in the logic level, and it does not define a new physical communication, letting *JSensor* to use the default `UnitDiskGraph` model.

- The Flooding application implements a simple `Message` that contains the sender and destination nodes, the message to be transmitted and a counter to hold how many hops the message had until the destination. In this case, a message containing sender ID, destination ID, a counter for the number of hops and the message content.

- The flooding application adopts a timer to send the initial messages (`MessageTime`). At the method *fire()* of the timer, the simulation sends the message to all neighbors of the node via broadcast. To start the timer, its method *startRelative()* is called, passing as argument a time and a node.

- The Flooding application adopts the `ReliableDelivery` model, which does not drop any message.

  There is also the `LossyDelivery` model in *JSensor*, which drops 5% of messages randomly.

- The Flooding application adopts `ConstantTime` transmission model, which returns a constant time.

  JSensor has also the `TimeByNode` transmission model, which returns a value based on the type of node sending the message.

- *JSensor* has the `ClimateEvent` model used by the Mobile Phone simulation to change the climate according to previously defined states, precisely sunny, cloudy, rainy and rainy with lightning.

- On Mobile Phone application, the climate event is distributed in grid implemented by `GridDistribution` model, so it covers all the environment.

- The Air Quality application uses a `Cell Model` with CO_2 level, water level, and a boolean value for lightning presence through the `AirConditionCell` model. The CO_2 levels indicate the amount of gas in a cell. The water level means the water accumulated by several hours of rain. The lightning indicates if it struck a cell recently.

- The Mobile Phone application adopts the `Set Event Cell` to deactivate nodes struck by lightning, changing the node state to disabled.

- The Air Quality application uses the `IncreaseCO2` model to increase the CO2 level based on the node that produced it.
- *JSensor* has `Random Way Point` mobility model to guarantee that a node will move in one of eight possible positions selected randomly or will stay in the same place an aleatory time. The user in the simulator can also define the distance traveled by the nodes in each movement. All applications implemented use the `Random Way Point` model.
- All simulations use the `CustomGlobal` implementation. This class is especially useful for debugging and obtaining global simulator information, so, for instance, we can use *JSensor*'s logging files at the end of each round to write which cells had CO_2 level above a threshold, using the *postRound()* method. The simulation uses the method *postRun()* when it finishes, for example, to write the final values of the CO_2 level from each cell.

Besides the above implementations, *JSensor* provides two fundamental routing algorithms: DSR Johnson et al. (2001) and AODV Perkins et al. (2003). The simulation could straightforwardly integrate these algorithms. We use these routing algorithms in the Mobile phone and Air quality applications.

3.5 *JSensor* Parallel Kernel

JSensor simulates nodes and application-events in parallel over multicore HPC architectures, as illustrated by Figure 3.3. The `Runtime` class, introduced previously, manages a pool of worker threads, where each worker is responsible for one or more chunks. As depicted on Figure 3.3, each chunk contains the respective set of nodes and application-events. In summary, there are w workers active in a machine with w cores and each worker has a similar number of chunks and nodes to simulate.

The nodes share a grid of cells, named environment, and each *JSensor* cell has coordinates latitude-longitude and which nodes exist in the cell. Cells exist only if there is at least one node dwelling it or if an application-event has modified it. As shown in Figure 3.3, the worker threads and, consequently, the nodes access the grid to change properties of the environment, so for concurrence reason, the grid needs to be thread-safe.

The size of the cells to build the grid can be varied. Thus the grid size parameter impacts the runtime of the simulations since the simulator needs to reach more cells to define the neighborhoods of a node when cells size is small. We identify the same effect when we use large communication ranges in the node.

The Figure 3.4 illustrates three examples of a grid with three different cell sizes. The first grid represents a scenario where the node radius is 300 meters, and the cell size is 150 meters. In this case, the number of cells within the communication radius of the node for which we want to define the neighborhood is high. The simulation does not check the cells

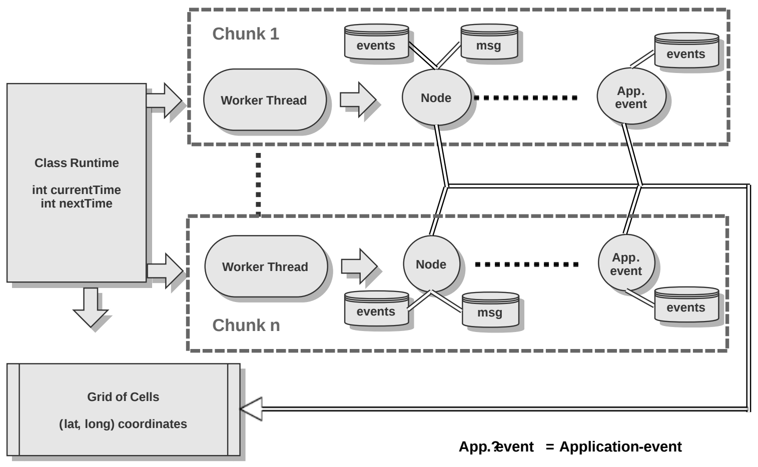

Figure 3.3: Parallelism in *JSensor*

that are entirely within the communication radius because the nodes that belong to them are neighbors. However, there are several partially covered cells, and therefore all nodes inside these cells should be checked to determine whether or not they are neighbors.

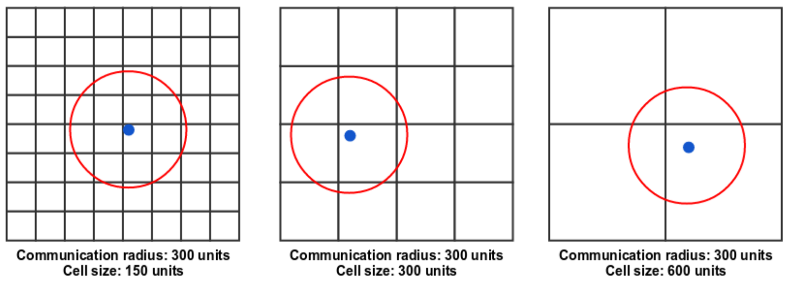
Communication radius: 300 units
Cell size: 150 units
Communication radius: 300 units
Cell size: 300 units
Communication radius: 300 units
Cell size: 600 units

Figure 3.4: Cell size influence.

As observed, more cells to be verified increase the runtime and the optimal cell size is a value near the communication radius of the nodes, as illustrated in Figure 3.4 by the second grid example, where the cell size and the communication radius are both 300 meters. The third scenario illustrates an example where the communication radius is 300 meters, and the cell size is 600 meters. In this case, once the cell size is bigger than the communication radius, the environment will have fewer cells. Although there are fewer cells to be verified, the number of nodes in each cell is significant, including also nodes far away from the communication radius of the node under evaluation.

Mobility events cause a drawback in the workload and, consequently, in the runtime. The nodes inhabit the cells, so when a node moves it can still stay in the same cell or move to a neighboring cell. The first case is simple because the simulator does not need to handle the cells, just changing the position of the nodes. If the node moves to another cell, *JSensor* creates a new one if it does not exist, due to on-demand creation strategy explained before, and then move the node. As a parallel kernel, several worker threads can be moving nodes to the same cells, so *JSensor* needs to ensure synchronization. Each worker thread blocks the cell until it finishes the write operations. The synchronization order to access a shared simulation resource, e.g., a cell or a node message inbox follow a predefined rule to achieve both performance and reproducibility, explained in details in the next section.

A significant capability is the synchronization strategy on *JSensor*. To perform the synchronization the class `Runtime` is responsible for maintaining events ordered according to its internal clock. *JSensor* creates synchronization barriers after the simulation of a new set of nodes in specific event time, so it assumes that there are nodes simulating events at the same time. Thus, they can execute in parallel without interference. It is important to make it clear that JSensor works with a bag-of-tasks style parallelization, where the elements of the simulation are split between the chunks to run in parallel.

JSensor goes further, detailing what execution means, so internally it enables a node to communicate, calculate its new neighbors, sense the environment, move through the simulated environment and so forth. This way, *JSensor* implements synchronization barriers, figure 3.5, after each simulation step finishes, i.e., after mobility, connectivity, send-receive message and sensing simulation steps, for instance. We simulate in parallel nodes operating at the same time and doing the same event type (e.g., sending a message), so it is possible to affirm that in *JSensor* there are no nodes in parallel at same simulation time and on different simulation steps. We adopt the same assumptions for application-events.

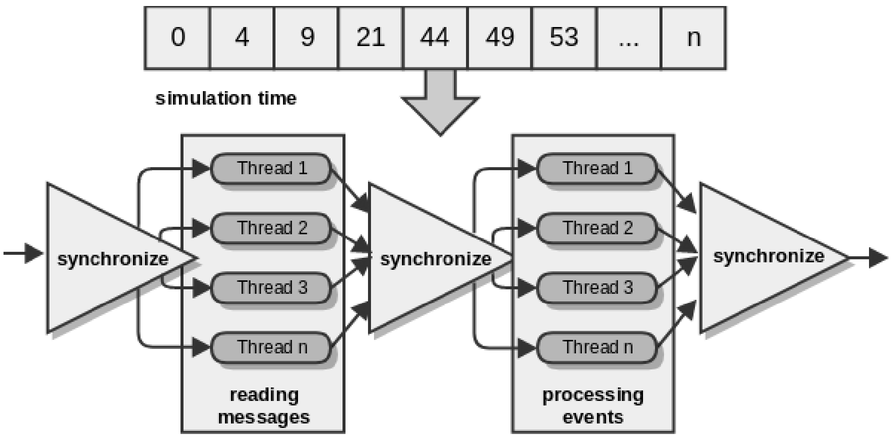

Figure 3.5: Synchronization barriers

Another significant capability of *JSensor* parallelization is that it does not adopt node localization to partition the workload, i.e., worker thread simulates any node of any region, so it tends to keep the workload fair among the workers. Workload strategies based on node position are often inefficient since nodes are not distributed uniformly in the environment once they move in the simulated area, changing their locations during the simulation. The partition of workload using protocol stack layers, provided in Zeng et al. (1998a); Doerel (2009) simulators, cannot scale for large number of threads, requiring a second task partition strategy.

Finally, the Algorithm 1 presents the core logic of `Runtime` class. It has access to all elements of the simulation, including *current_time*, the list of nodes N, the list of events E, and the shared grid in which the nodes reside G. The list of nodes N consists of k sublists, where k is the number of machine's cores. The same is true for the list of application-events E. This way, each sub-list Nk and Ek is associated to a chunk. This partition schema into chunks is required to guarantee that each thread simulates its respective nodes and application-events, avoiding concurrent accesses and, consequently, bottlenecks.

Algorithm 1 `Runtime` main execution

```
Require: $time$, $N$, $E$, $G$
 1: while $t \leq time$ do
 2:     for $i \leftarrow 0 \dots |\sigma|$ do
 3:         for $j \leftarrow 0 \dots k$ do
 4:             $events \leftarrow$ Hold the list of events will execute at time $t$
 5:             Performs the $events$ of the state $\sigma_i$ of all nodes
 6:             $events \leftarrow$ Hold the list of events to be executed at time $t$
 7:             Performs the $events$ of the state $\sigma_i$
 8:         end for
 9:         Synchronize all $N$ who performed $\sigma_i$
10:     end for
11: end while
```

Analyzing the Algorithm 1 we have:

- **Line 1**: Loop definition to execute each simulation time t.
- **Line 2**: Loop definition to execute all simulation steps σ. The steps are partitioned according to actions of the simulator, such as: σ_1 mobility, σ_2 connectivity, σ_3 message send-receive, σ_4 sensing, and σ_5 general-purpose computing events.
- **Line 3**: Loop definition to execute all k threads. In each σ step the simulator starts all k threads of the simulation with their list of nodes and application-events, executing all events that have compatible time. More specifically, we implemented a thread pool to avoid the creation and destruction of threads repeatedly, thus maintaining some threads active while the simulation occurs.
- **Line 9**: Between two steps there is a synchronization barrier to prevent threads from different steps to be started until all threads in the current step have finished their step

execution. This strategy ensures that nodes or application-events can move, connect, send-receive messages and create new events in parallel, however, one step at the time.

3.6 *JSensor* Reproducibility

Parallelism affects both simulations causality, creating changes over events dependents, as the numerical reproducibility. The events can be simulated in a different order during different runs of the same simulation because of the non-reproducibility of communication between processors. The change in the order of only two events of a simulation may end up generating entirely different results. As detailed in Cleveland et al. (2013b), the change of event simulation order can lead to numerical inconsistency, caused by changing the order of double precision floating operations. This problem occurs not only with mathematical operations but in any simulation that deals with non-commutative operations.

As a simulator, *JSensor* must allow the reproducibility of simulations. In Dalle (2012b), the authors detail types of reproducibility, as well as simulations examples and reproducibility requirements. There are four levels of reproducibility, level one is the most restricted and used by *JSensor*. Level one is also known as the repeatability property; it designates the ability to re-execute the same simulation history as previously regarding computation, including the inherently non-deterministic computations. In the case of discrete-event simulation, for example, repeatability means that the simulations produce the same series of events and process them in the same order. In the case of distributed simulations, in particular, repeatability implies that parallel activities, e.g., when the event occurs at the same simulation time, the processing must always be in the same order, which requires some additional synchronization Fujimoto (2000).

Reproducibility of sequential simulations is trivial because they need a seed and a good random number generator. In this way, we ensure the same sequence of numbers from the seed as often as necessary. In case of parallel simulators, the reproducibility is not trivial, since multiple threads or processes execute concurrently and often nodes are simulated without any synchronization. *JSensor* has reproducibility level one of the sequential and parallel simulations, regardless the hardware used or the number of threads started to perform the simulation. A simulation can contain non-deterministic models, arising from the use of pseudo-random numbers as alternatives for defining probabilities. Algorithms for mobility, communication, the initial distribution of nodes and many more frequently use pseudo-random numbers.

JSensor implements the concept of chunk to allow reproducibility, even when executing non-deterministic simulations in parallel. A chunk encapsulates a set of nodes and application-events, a seed to ensure a starting point to generate pseudo-random numbers and a generator of pseudo-random numbers. The number of chunks can be the number of machine processing cores or any other positive integer number. If we consider the number of machine processing cores to define the number of chunks, *JSensor* allows the user to start it with worker thread pool equals or less than the number of machine cores, so it ensures reproducibility of simulations with the same amount of threads or not. In summary, we adopt the highest concurrency level to define the number of chunks, since each worker thread must manage at least one. This way, all other concurrence level adopted in the same machine or not will be smaller and repeatable by *JSensor*.

As explained above, each worker thread simulates a set of nodes and application-events. Therefore, each thread can simulate one or more chunks of nodes and application-events. The chunks are never shared among threads to avoid unnecessary synchronizations. The idea to start from only one seed and to generate other seeds based on the number of chunks enables reproducibility in parallel since now each chunk with several nodes and application-events have their seeds to repeat in other executions. Each chunk has a seed and a random pseudo-numbers generator, regardless the number of threads on execution. Non-proportional numbers to define both the number of worker threads and the number of chunks, e.g., four workers and seven chunks, in a simulation can allocate more chunks to one worker thread than others, so they should be avoided.

Figure 3.6 illustrates a machine with four processing cores and how it is possible to compare the same simulations running with two threads and then with four worker threads. Initially, the nodes are partitioned into four chunks, regardless of the number of threads the user selected in the simulation. In Figure 3.6 *(a)*, the user chooses two threads. Thus, each thread will simulate the nodes and application-events of two chunks sequentially. In the Figure 3.6 *(b)*, we have four threads, so each thread simulates the nodes and the application-events of a single chunk. The *JSensor* simulates nodes and application-events of a chunk in the same order, and we guarantee the same sequence of pseudo-random numbers from a seed and a generator since each chunk is a sequential simulation. Thus, as shown in Figure 3.6, no matter the number of worker threads, each thread *JSensor* always simulate the same nodes and application-events in the same order. The unique restriction is that the number of chunks requires being higher or equal than the number of worker threads.

When simulating cell properties, the users often use mathematical operations and other non-commutative operations. To maintain the reproducibility, we store or repeat the events in the same order, before being simulated. In Robey et al. (2011b), the authors mention that sorting technique is a distinct method to obtain numerical consistency. Besides the advantage of simplicity, this technique also has the benefit of being generic, since we are not seeking only the reproducibility of mathematical operations, but any activity in cell properties. The simulator partitions the cells into chunks, and the kernel ensures that the simulation accesses each chunk of cells through only one thread. As mentioned before, this strategy avoids synchronization, ensuring scalability. The simulator stores and sort the grid of cells into chunks. Thus, we achieve for cells the same reproducibility level assumptions for nodes

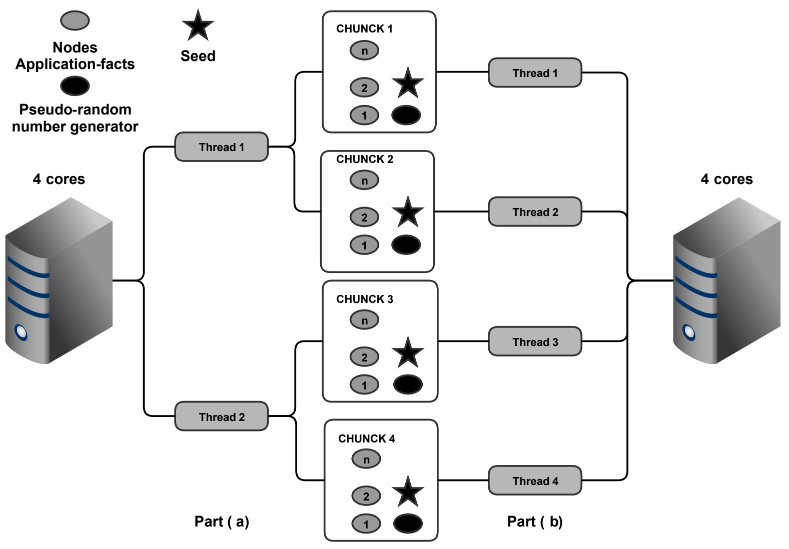

Figure 3.6: Reproducibility using chunks in *JSensor*.

and application-events.

3.7 Available *JSensor* Applications

In this section, we explain the available applications in *JSensor* used to perform the evaluations explained in the next section.

3.7.1 Flooding Application

In a data propagation application, like flooding, it is reasonable that all nodes are interconnected, and all messages can propagate to all nodes with the same probability. In *JSensor* flooding application, nodes send messages copies to their neighbors, that consists of all nodes inside the sending node radius.

To start the message propagation, the model selects nodes to send initial messages. Each of this nodes must create a new message and select one node of the network as a target. When a node receives a new message, it compares its ID with the message target ID. If it is the target, reports the receipt of the message and finishes the propagation of this message copy. Otherwise, the target forwards the message to its neighbors. The simulation runs until all reachable nodes have received all messages.

This simulation presents few code lines, but it stresses a simulator regarding communication, generating a massive number of messages. We tested few different scenarios using *JSensor*, Sinalgo, and OMNeT++. In the following models, we fix the simulation area in 84.000 square units, and we position the nodes randomly. To start the simulation ten initial nodes are selected to send the first messages and the communication radius is 300 units. Besides, other features are added to the models for each test scenario, as presented in Table 3.2.

Table 3.2: Features added for each scenario

Scenario	#nodes	net. density	mobility?	cells size
#1	125K to 1M	5 to 40	NO	300 un.
#2	500K	20	0 to 100%	300 un.
#3	500K	20	100%	75 to 300 un.
#4	2.4K to 38.4K	2	NO	300 un.

When we compare results from the different simulators, we face a significant challenge, because the results will hardly be the same. That happens due to the different decisions made by the kernel during the simulations. For example, the order the simulator executes the events at the same time can lead to results substantially distinct.

To make a fair comparison between *JSensor* and OMNeT++, we used a scenario where the results are equals. Making some adjustments in the flooding model, we used the scenario 4, that exchanges a massive quantity of messages among nodes, always keeping its path the same, independent of the simulator used. The network consists of a ring topology, and the application always send messages to a node with a position in the ring equals $sn + \lceil n/2 \rceil + 1$, where sn is the node sending a message, and n is the total of nodes. Using this model, one of the paths of the ring will always be shorter, avoiding different results that have the same cost. All nodes in the network send a message that is transmitted node by node until it reaches the destination.

3.7.2 Mobile Phone Application

The mobile phone application aims to test the scalability, exploring new scenarios not used in the Flooding model. It considers mobility models, messages loss, routing algorithms, distribution models, different message delays, and natural phenomena, making a more realistic simulation. These new models also increase the simulation workload, making it possible to evaluate the simulator performance running more complex simulations.

The model mimics a mobile phone network with antennas distributed in the simulation area based on a grid distribution, aiming a better signal coverage and they cannot move through the area. The mobile phones' distribution is random, and 30 to 60% of them move using the `RandomWayPoint` model presented in Section 3.4. The base station location has to be near the mobile phone users to provide an acceptable reception quality. The mobile phones can move through the environment and made calls to other ones through the Antennas. One of the factors that influence the signal quality is the weather. To simulate this, application-events like rains and lightning occurs from 20 to 30 time units and their incidence in a cell where an Antenna lives can inactivate it, forcing the mobile phones to find other routes.

There are 20,000 cell phones and 25 antennas. The communication between nodes is made using messages. When the simulations start 30 to 60% of mobile phones, they send an initial message aiming another mobile phone, but they cannot communicate directly. The message needs to pass through the Antenna that provides the signal for this mobile phone. For Antenna-Antenna communications DSR routing algorithm is used, and the message has to arrive the Antenna where the destination mobile phone is connected. Every received message by a mobile phone has 60% chance of being replied, keeping the data exchanging between the mobile phones.

Mobile phones and Antennas have different communication reliability, so it is normal that the number of lost packages to be different. The `LossyDelivery` used defines that 1% of messages send by Antennas will be lost, and 3% for the mobile phones. The transmission time for each message is also different, so Antennas spend 1 unit of time to deliver a message to its destination and mobile phones 2 units. The models used on flooding and mobile phone experiments take the simulator to the extreme concerning messages. Besides, other features are added to the models for each test scenario, as presented in Table 3.3.

Table 3.3: Features added for each scenario

Scenario	mobility?	Appl.-event	messages
#1	30%	30	30
#2	45%	25	45
#3	60%	20	60

3.7.3 Air Quality Application

The air quality simulation mimics the carbon cycle in the environment. The simulation nodes are classified into CO_2 producers and consumers. Trees and rain reduce carbon taxes from environment cells. We classify trees into three types: big, medium and small trees. We define the CO_2 absorption according to tree size. Animals, including humans, produce CO_2. Machines like cars, trucks and many others, as well as industries, produce CO_2 in different rates. The total number of nodes is 190,0020, where 20 industries, 5,000 houses, 20,000 humans, 10,000 cows, 100,000 small trees, 50,000 medium trees, 1,000 big trees, 3,000 cars and 1,000 trucks.

To make the model more realistic, we adopted a random behavior guided by lower-upper bounded frontiers, i.e., caws producing 1-5% of CO_2 when compared with a car and a car producing 1-5% of CO_2 when compared with industry. The same idea is adopted to simulate nodes initial position, event lifetime, event size, node-event mobility and many more. The air quality simulation has approximately 1,300 lines of code and 21 customized *JSensor* classes, so it is the more complex one. The complexity here means the number of nodes and application-events, besides the variety of types and the lifetime event that mimics death and born of nodes, like trees, humans and so forth.

We model the environment as cells with size 500×500 units, and properties CO_2, water and lightning, where CO_2 and water are double values and lightning is a boolean value. If a cell has water level above the threshold, it has a probability to become off by flood or by a lightning incident (value true). The consequence is all nodes living in the cell must be off for a period in the simulation, and then they must restart working. The idea is to mimic rain and lightning occurrences, a moment off to repair malfunctions and a moment when nodes start working again.

The initial position of nodes (cars, trees, and industries) are aleatory, as well as humans and caws on random initial locations and walking through the environment. Human, caws, trucks, and cars have a mobility model which assumes the space as eight possible regions to move a defined distance. All mobile nodes share the same mobility model, but with different step distance to each node type, e.g., a caw and a car have different ranges for each movement, since they walk in different speeds. This application considers only one scenario; thus, it was not necessary to use features variations.

3.8 Experiments

To perform the simulations, we use two different machines: Machine 1 (M1) equipped with 2 Intel Xeon processors E5-2620, each processor with six physical cores and six cores with hyper-threading technology, both operating at 2.0 GHz each core and 128GB of RAM DDR3 1333Mhz. The Operating System was an Ubuntu 14.04.5 LTS 64 bits kernel 3.13.0-123-generic; and Machine 2 (M2) equipped with 2 Intel Xeon processors E5620, each processor with four physical cores and four cores with hyper-threading technology, both operating at 2.4 GHz each core and 48GB of RAM DDR3 1333Mhz. The Operating System was a CentOS 7 64 bits, kernel $3.10.0 - 693.21.1.el7.x86_64$. In M1 and M2 all experiments could fit in RAM.

We repeat each experiment five times, and we calculate the average time from these runtime values. We adopt the same strategy when memory consumption is measured. We implement all *JSensor* algorithms in Java 64 bits (JAVA 8 - update 45).

3.8.1 Flooding Application Experiments

In this section we perform two comparative tests: *JSensor* versus Sinalgo, and *JSensor* versus OMNeT++. Besides comparative results, in this section it is presented *JSensor* overheads when the mobility and the cell size increase.

To make comparisons we configure the simulators to log the same information in the simulations, so it possible to verify reproducibility. In the flooding model, it was recorded the initial messages and the messages received by the destination nodes. All logs contain information about the simulation time the communication event occurred, precisely the node that started the message, the target node of the message, the message number of hops, and which nodes form the message route.

3.8.1.1 *JSensor* versus Sinalgo

JSensor models are based in Sinalgo simulator Distributed Computing Group (2015). Thus, we used it to evaluate the scalability. We use the machine M2 in the Flooding scenario #1, described in Section 3.7.1.

Figure 3.7 shows *JSensor* and Sinalgo runtimes when the network size increases in the simulation. The network size indicates how many nodes the simulation runs. We tested both with 125K to 1M nodes, using sequential Sinalgo and *JSensor* with one and sixteen threads. *JSensor* with sixteen is faster than Sinalgo in all experiment instances, precisely near 9 times units faster when 1M of nodes are simulated. The one thread execution has almost the same runtime than Sinalgo in all experiments, what means that HPC *JSensor* features did not deteriorate its execution too much. In summary, Sinalgo runtime deteriorates more than parallel *JSensor* when nodes increase.

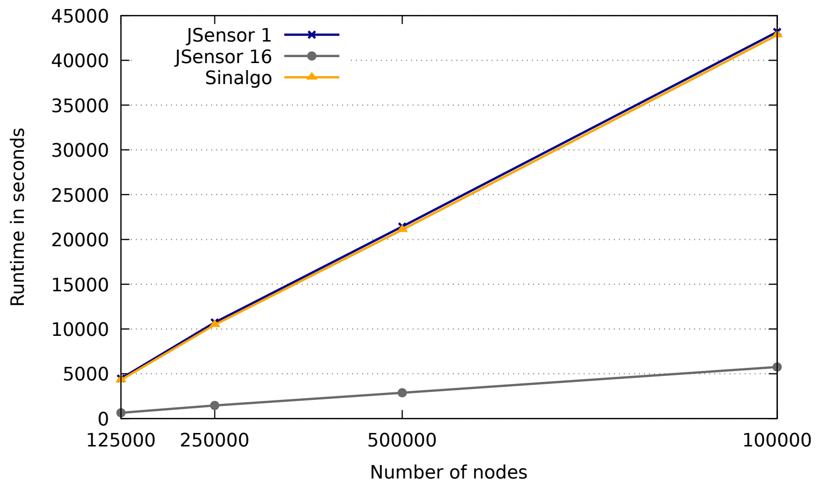

Figure 3.7: Runtime *JSensor* versus Sinalgo.

When we increased the size of the network, the difference between the simulation times became expressive, reinforcing the HPC advantages of *JSensor*. It happens because the events in the same step have a high level of independence so that they execute in parallel. The parallel kernel portion consumes part of the computational resources, what makes the difference in simulating small and huge scenarios. When the simulation becomes harder, the parallelism increases so that the kernel can achieve an almost reasonable workload, what makes a huge difference in the total simulation time.

It is important to reinforce that in all tests the number of messages and all application features were the same in all simulations performed in both simulators, corroborating with *JSensor* reproducible characteristics.

3.8.1.2 ***JSensor*** **versus OMNeT++**

We test the Flooding scenario #4 with the machine M1 in parallel and using the maximum of available cores, i.e., in both simulators it was considered 24 threads. The results presented in the Figure 3.8 confirms our hypothesis that *JSensor* performs well for large simulations. For the 9,600 nodes *JSensor* presented a runtime 0,2% slower than OMNeT++, but as the number of nodes increased the difference in runtime became expressive, reaching 21% faster than OMNeT++ when 38,400 nodes were simulated.

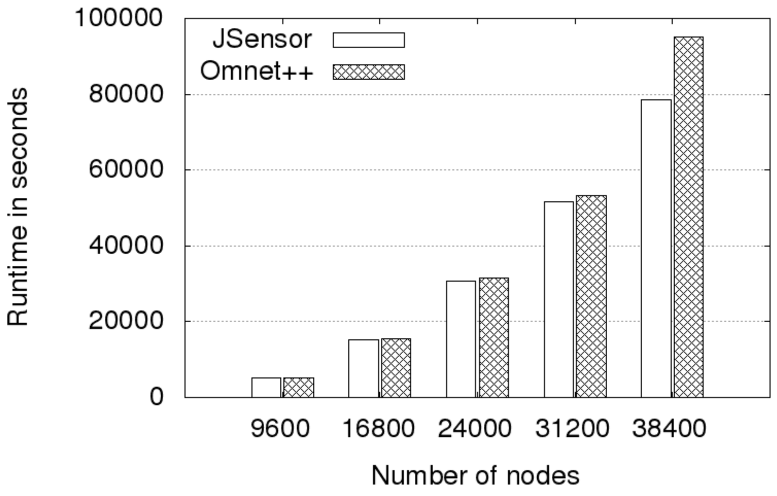

Figure 3.8: Runtime *JSensor* versus OMNeT++

The reason why *JSensor* presented better results for large simulations is that we design its entire kernel for concurrent computing. Further, the use of the cells grid allows the faster node neighbors discovery, the deactivation of modules not used by the simulation and the use of thread pools save simulation time. The promising result is not only 21% faster in large simulations but also be 0,2% slower in small simulations, demonstrating the low overhead introduced by the native parallelism of *JSensor*.

In terms of memory consumption the difference between simulators was considerable, as Figure 3.9 illustrates. The most significant difference occurred in the instance with 9,600 nodes, being *JSensor* memory consumption 7.6 times greater than OMNeT++. The positive aspect is that as the number of nodes increases, the difference in memory consumption decreases, reaching 3.5 times more memory than OMNeT++ with 38,400 nodes. The main reason for this behavior is that *JSensor* creates copies of messages, cells and several other WSN abstractions to guarantee safe write operations and the reproducibility, so in this case, *JSensor* introduce memory overhead. A Java element is the garbage collector, which runs periodically, but in a non-deterministic way for Java developers, so the memory consumption is always measured in a fuzzy way. Finally, *JSensor* operates over a JVM that introduces its memory overhead.

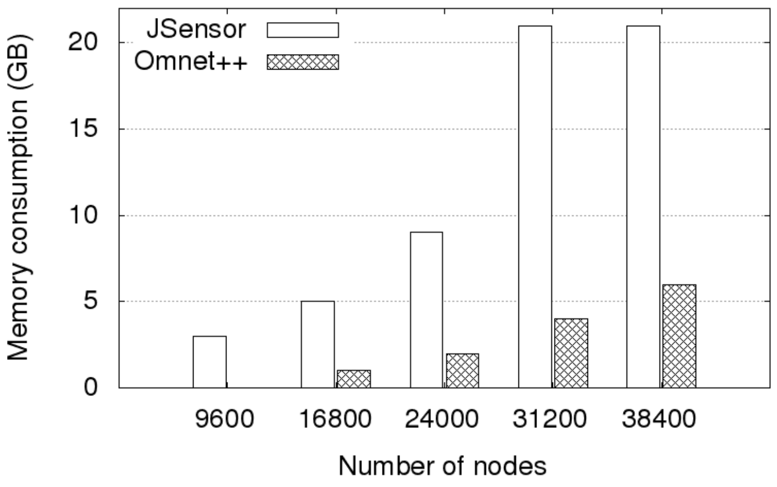

Figure 3.9: Memory Consumption *JSensor* versus OMNeT++.

All tests performed in this section produced the same messages in both simulators using their logs. These results corroborate with *JSensor* reproducible characteristics.

3.8.1.3 *JSensor* Speedup Evaluation

Once we have the simulation times of *JSensor* and Sinalgo (see Figure 3.7), we can calculate the speedup of *JSensor* when we increase the number of threads. Sinalgo was selected as the best sequential simulator, since it is faster than *JSensor* with one thread. *JSensor* uses sixteen threads and the experiment considered 1M of nodes flooding messages during several hours per simulation. Each flooding simulation produced around 10M of messages and the simulation finishes when the nodes handle all messages. We calculate the speedup as $speedup = s/p$, where s is the best sequential simulation found and p is the parallel one. Figure 3.10 illustrates that *JSensor* achieved a maximum speedup of 7.45 times in a machine with sixteen cores, where only eight are physical ones.

Initially, this result sounds not a very good speedup one; however, it is using Amdahl's Law or its predecessor Gustafson Law. Amdahl's Law Amdahl (1967) made an observation: "the serial part of the program limits the speedup, and the curve represents a diminishing return.". This observation means that a simulation with a 90% parallel portion could achieve a speedup of 6.4 using 16 processors and around 10 regardless of the number of processors used. The input instance size changes the speedup, so better speedups can be achieved as the input increases, but not forever, so Gustafson Gustafson (1988) is more complete, but it is also unrealistic for some domains.

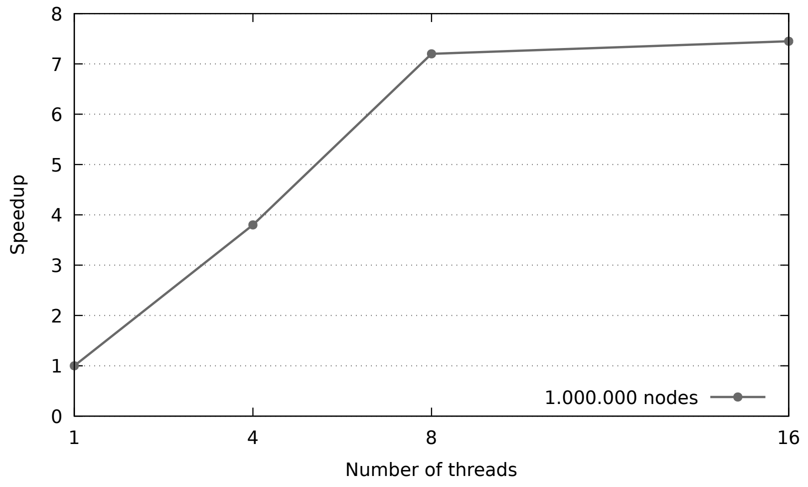

Figure 3.10: *JSensor* speedup

Using Gustafson or Amdahl and being conservative, a speedup of 7.45 indicates that the simulation has a parallelization near 90%, which means very promising since flooding is communication intensive. Typically, several nodes are sending/receiving messages at each simulation event, enabling *JSensor* to execute them in parallel. All the tests performed in this section have the same number and type of messages and nodes, corroborating the reproducibility with any number of threads.

3.8.1.4 Mobility Overhead

To check the effect of mobility in the *JSensor* simulations, we performed several tests using the Flooding scenario #2 and the machine M2. First, we set the number of nodes in 500K and vary the percentage of nodes with mobility between 0 to 100%, as shown in the Figure 3.11. *JSensor* introduces a significant overhead when nodes are mobile, but the increase in the number of mobile nodes does not affect its runtime drastically. With 0% of nodes with mobility it spends 3K seconds and if we increase the mobility 25 times the runtime increased only 2 times, becoming 6K seconds. After 25% of mobility, the impact in runtime reduced drastically, becoming residual, precisely varied from 7K seconds to near 9K seconds with all nodes mobile.

JSensor only creates necessary structures if there is at least one node with mobility, what explains the fast behavior of flooding without mobile nodes. The memory consumption did not present any relevant variation with the variation of the number of mobile nodes. All simulations performed in this section maintain the number and type of both nodes and messages identical.

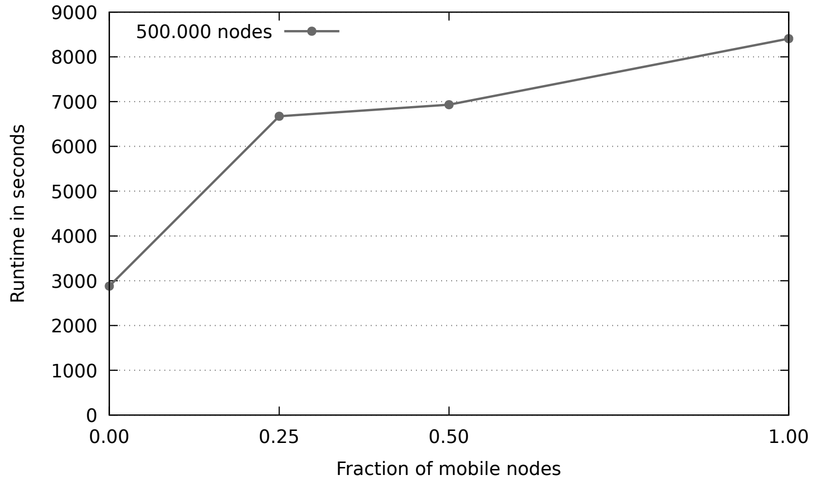

Figure 3.11: Mobility performance.

3.8.1.5 Cell Size Overhead

We also conduct tests to check the impact of the size of the cells in the simulation time and memory usage using the Flooding scenario #3 and the machine M2. Figure 3.12 illustrates the runtime results when the cell size varied in a network with 500K mobile nodes. The tests consider only mobile nodes because mobility cost is influenced by the cells sizes, as explained before.

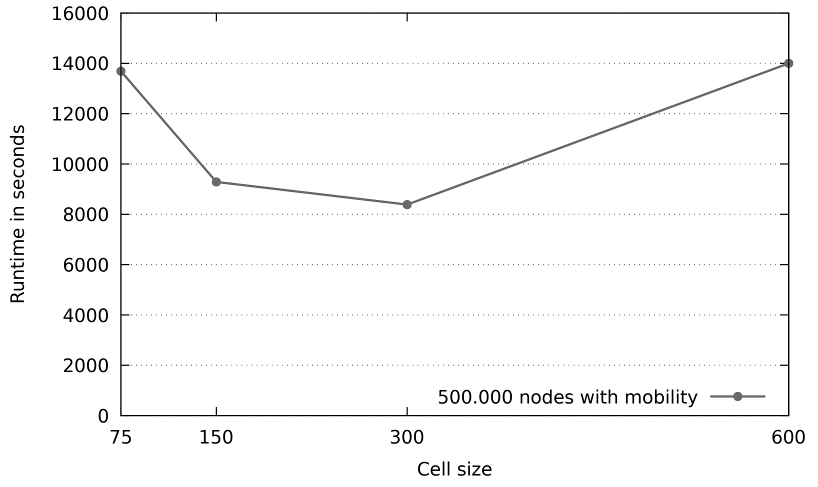

Figure 3.12: Performance versus cell size.

As depicted in Figure 3.12, if we double the cell size, the execution time increased more than 67%, so these results confirm the hypothesis presented in Section 3.5. The size of the cell with the best performance among the tested scenarios is the one with the same size as the nodes communication radius, precisely 300 units in Figure 3.12. In this case, the number of cells to be scanned is small, and cells entirely inside the communication radius do not need to have their nodes checked, saving CPU clocks. Cells that are not entirely within the radius must have all nodes checked to make sure that they are neighbors.

Although the variation of memory consumption is not alarming, as presented in Figure 3.13, we found that using smaller cells ends up consuming more memory. We expect this result because the more cells to store more memory is needed. All simulations performed in this section maintain the number and type of both nodes and messages identical.

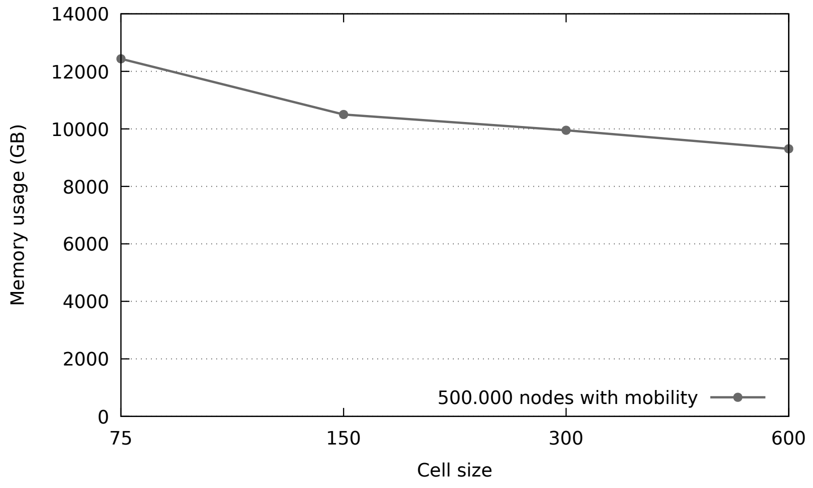

Figure 3.13: Memory versus cell size.

3.8.2 Mobile Phone Application Experiments

In flooding tests, we have used straightforward communication models to make possible to compare with other simulators. In this section, we explore more features to check up *JSensor*'s behavior in more complex scenarios. We have tested the mobile phone model using three scenarios, as described in Section 3.7.2, to evaluate the impact of increasing mobility, nodes, application-events and messages in a simulation. Figure 3.14 shows a similar speedup behavior for the three scenarios, with a small variation in the results (from 6.8 to 7.4). The scenario #3 simulation with the best speedup result can be explained by Gustafson discoveries Gustafson (1988) since it was the larger than scenario #1 and #2 simulations regarding the number of messages, application-events occurring more frequently and more nodes mobile.

The impact of scenario #3 concerning memory consumption was low when compared with other simulations in Figure 3.15. In general, the parallel versions using 6, 12, 18 and 24 threads varied the concurrence level in 4 times, but the memory consumption ranged less than 10% from 6 to 24 threads, demonstrating a promising result. The difference between single-thread version and multiple-thread version, precisely with 6 threads is considerable since there are concurrence overheads in the parallel version. In Figure 3.15 the memory consumption in-

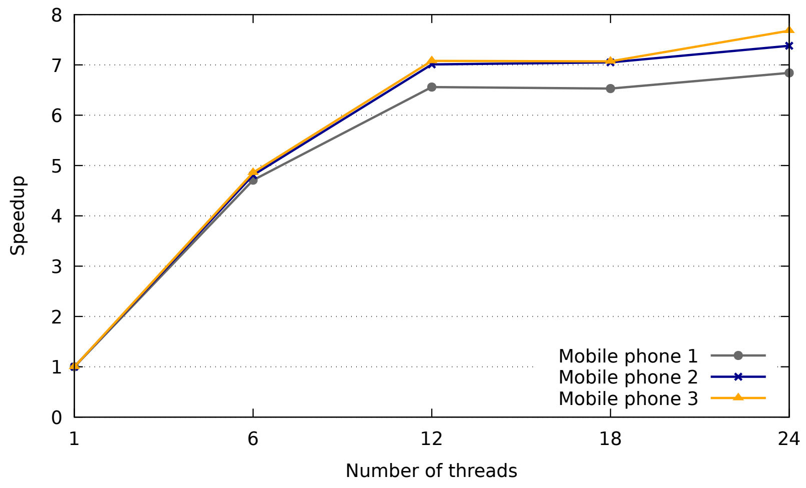

Figure 3.14: Mobile phone application speedup.

creased from 3GB to near 11GB or more than 3.5 times when we compare the one thread version with the six one. Internally, *JSensor* uses data structures and algorithms with the faster runtime, i.e., with low complexity concerning runtime, being memory consumption a secondary requirement, thus not fundamental. This way, the concurrent multi-thread executions tend to consume much more RAM than single ones.

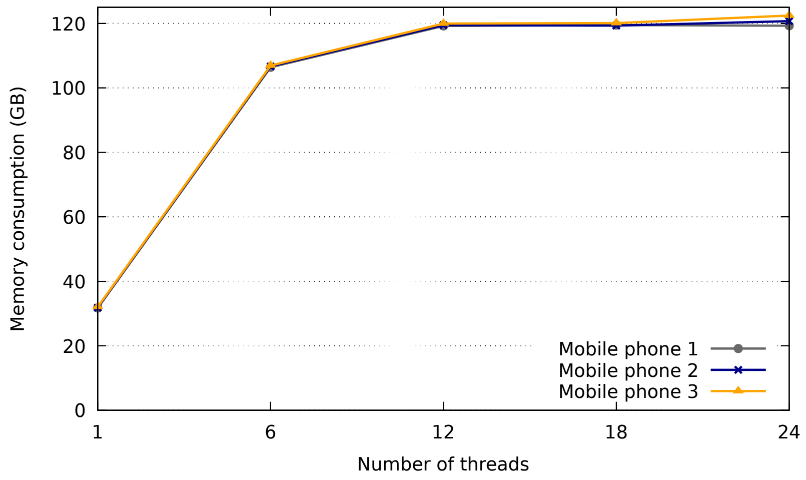

Figure 3.15: Memory consumption.

In the mobile phone model, we recorded which were the initial messages sent, the messages received by the destination or target nodes, the application event and mobility events. The message logs contain information about which simulation time the event occurred, which node sent it, which target node, how many hops, and by which nodes the message has passed. The Application event logs inform if a cell has been struck by lightning or is flooded. Mobility events contain information about the node's previous position and what the current position is. Comparing these files we verified the reproducibility of the simulations. As in Flooding application (Section 3.8.1), in all executions the number of messages and all application features were the same in all simulations performed.

3.8.3 Air Quality Application Experiments

To explore more the application-events, environment modeling, timers and mobility features we test the air quality scenario using the machine M1. Figure 3.16 presents the speedup of air quality scenario, where with 24 threads the simulation achieved a speedup of 5.19, so as the simulation complexity increases the speedup decreases, an expected behavior since the synchronizations and object copies, for instance, increase. In these experiments, the amount of CO_2, the water level and if a cell is off of operation represent modifications in the environment that must maintain the reproducibility, so they are very often CPU-bound.

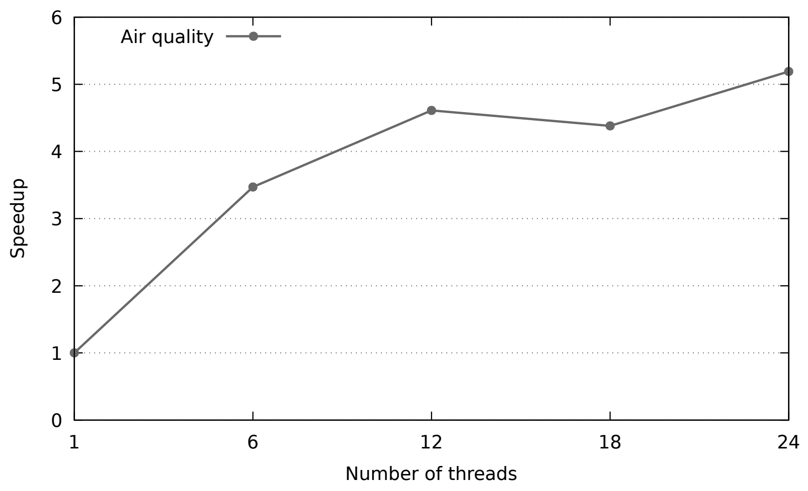

Figure 3.16: Speedup air quality application.

Another feature widely used in this model is mobility. In air quality application all the humans, cows, cars and trucks move during all the simulation. When a node moves through the environment, it can stay in the same cell or change the cell. In the first case, the kernel only needs to change the node position in the cell, but if the node changes the cell, the kernel can have to create a new cell, if it does not exist, remove the node from the old cell and insert it in the new one. All these other operations to guarantee mobility is also time-consuming, thus influence negatively in speedup.

However, if we consider a speedup above 7 as a simulation with 90% of parallelism with 16 processors, the air quality application with speedup above 5 can indicate parallelism of 80% with 24 processors, an also promising result, especially if we consider the complexities introduced in the simulation. The memory consumption did not show significant variation among the experiment instances, maintaining an average of 127GB of RAM used by each simulation.

The air quality model contains the same log information already presented in the flooding and mobile phone models: messaging log, mobility, application events and so forth. All information is recorded for each type of application-event and nodes. We verified the logs, reaching identical results, this way corroborating with the reproducibility even when different number of threads are configured.

3.9 Modeling Limitation

We expose several *JSensor* benefits and drawbacks regarding its implementation and designs in previous sections. In Section 3.5, it was presented that the kernel has synchronization barriers after each step, where the order of the events is done to maintain reproducibility. The synchronization order to access a shared simulation resource follows a predefined rule, precisely *JSensor* uses the node, application-event, message or any other abstraction ID to perform the order, using them in ascending order. Then, the lower event ID implies higher priority. This strategy was implemented to reduce the runtime of message inbox checks, the grid of cells management, neighborhood calculus, and so forth, since sorted collections enable binary searches, logarithmic insertion/removal operations, and other optimal operations.

Unfortunately, the same order during multiple synchronization barriers in a single simulation is sometimes not realistic. If we suppose two cows are deciding to eat the grass of the same cell multiples times during a single simulation, it is neither fair nor realistic, that the same cow always wins, regardless the number of times they concur for the corresponding cell in the same simulation.

Unfortunately, this limitation is present in other HPC simulators, like those presented in Section 2.3. A solution is to support an aleatory order using the same seed used for other purposes in a sequential simulation per chunk. This way, it guarantees that concurrent events to move, send/receive a message and others will not obey a static order during a single simulation, but also preserving the reproducibility of multiple simulations. This realism will introduce overheads in simulation, since before processing a collection of messages or cells or nodes or any other simulator abstraction in single event time, the kernel will generate their ordering using a seed number, as mentioned before, creating many more insertion operations and data structures copies to guarantee dynamic order.

3.10 Conclusion

In this work, we present *JSensor*, a parallel Java simulator to large-scale WSN simulations and general purpose simulations. We highlighted the *JSensor* main features, we compared it with the literature, and a detailed comparative table summarizes the core differences and similarities of both *JSensor* and related work. Different simulation applications, i.e., communication intensive simulation with a flooding model, a WSN simulation with a mobile phone model with two routing algorithms, and we implemented a general purpose simulation for air quality and tested it exhaustively. In general, *JSensor* proved to scale very well for CPU-bound WSN or general purpose simulations with thousands of nodes and millions of messages. Cell size and mobility features did not affect the simulations drastically concerning memory usage and runtime, demonstrating that the presented solution scales well even in extreme scenarios.

JSensor future plans include: i) adoption of tiles, proposed in Sundresh et al. (2004), in conjunction with spatial cells; ii) implementation of other wireless and wired network protocol standards especially protocols for simulating MAC and physical layers; iii) implementation of a UI for large-scale WSN simulations based on a distributed publish-subscribe architectural style, where developers or any end-user can observe the simulations remotely; iv) implementation of a multicomputer version for *JSensor*; v) implementation of a multicore version for *JSensor* where graphical processing units (GPUs) are considered; and vi) implementation of the option for simulating dynamic ordering during synchronizations, as detailed in Section 3.9.

Chapter 4

Conclusion

In this work, we present the first generic reproducibility model to be used in kernel level by WSN simulators. This way, repeatable simulations can occur without users interferences. The model key ideas of how to partition the data and processing of a simulation were implemented and tested in *JSensor*, an existing HPC simulator useful for evaluating WSN applications and high-level distributed algorithms demands. Few steps were enough to introduce the most restrictive reproducibility level, known as the repeatability property; it designates the ability to re-execute the exact same simulation history as previously in terms of computation, including the inherently non-deterministic computations. Experiments demonstrated that the reproducibility model can be integrated in existing simulators and it produces correct results, even when users change the number of threads or the machines after each simulation run.

The second contribution of this work is a new version of *JSensor*, where reproducibility is implemented in kernel mode, environment modeling and application-events to simulate interferences from external agents or natural phenomena are also implemented in kernel mode, thus exposed to users in a simple, transparent and extensible way. An experimental evaluation demonstrated that *JSensor* scales very well in multi-core hardwares with a shared memory abstraction. Comparative evaluations reinforced that it is also a competitive WSN simulator for HPC since it outperformed OMNET++ while simulating a message intense flooding model. The results demonstrated that as the simulation enlarged in terms of number nodes, mobility level achieved, messages transmitted, environment cells occupied and so forth, *JSensor* versus OMNET++ difference in runtime also increased.

In contrast, when simulation scenarios are small the HPC designs of *JSensor* turn it slower than OMNET++ and its predecessor, the Sinalgo simulator. The main reason this happens is that the time spent in creating objects and performing data structure operations is high if compared to the time running small simulations sequentially. For these scenarios, it may be faster if you use *JSensor* configured with only one thread, rather than create multiple threads, split the simulation elements, and create synchronization barriers.

JSensor future plans include: (i) adoption of tiles, proposed in Sundresh et al. (2004), in conjunction with spatial cells; (ii) adoption of battery and energy models proposed in Chen et al. (2005); (iii) implementation of other wireless and wired network protocol standards, specially protocols for simulating MAC and physical layers; (iv) implementation of a UI for large-scale WSN simulations based on a distributed publish-subscribe architectural style, where developers or any user can observe the simulations remotely; (v) implementation of a multicomputer version for *JSensor*; (vi) implementation of a multicore version for *JSensor* where graphical processing units (GPUs) are considered; (vii) implementation of the option for simulating dynamic ordering during synchronizations. Besides *JSensor* improvements, we are interested in evaluating the reproducibility model with other WSN simulators to measure its integration impact in existing architectural designs.

Bibliography

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002a). A survey on sensor networks. *IEEE Commununication Magazine*, 40(8):102–114.

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002b). Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422.

Ali, R., Solis, C., Salehie, M., Omoronyia, I., Nuseibeh, B., and Maalej, W. (2011). Social sensing: when users become monitors. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 476–479. ACM.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA. ACM.

Angelopoulos, C.-M., Nikoletseas, S., Patroumpa, D., and Rolim, J. (2010). Coverage-adaptive random walks for fast sensory data collection. In *9th international conference on Ad-hoc, mobile and wireless networks.*

Aquino, A. L. L. and Nakamura, E. F. (2009). Data centric sensor stream reduction for real-time applications in wireless sensor networks. *Sensors (Basel)*, 9:9666–9688.

Bagrodia, R., Takai, M., an Chen, Y., Zeng, X., and Martin, J. (1998). Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, 31:77–85.

Barr, R., Haas, Z. J., and van Renesse, R. (2005a). JiST: An efficient approach to simulation using virtual machines: Research articles. *Software Practice and Experience*, 35(6):539–576.

Barr, R., Haas, Z. J., and van Renesse, R. (2005b). Jist: An efficient approach to simulation using virtual machines: Research articles. *Softw. Pract. Exper.*, 35(6):539–576.

Chen, G., Branch, J., Pflug, M., Zhu, L., and Szymanski, B. (2005). Sense: a wireless sensor network simulator. In *Advances in pervasive computing and networking*, pages 249–267. Springer.

Chhimwal, P., Rai, D. S., and Rawat, D. (2013a). Comparison between different wireless sensor simulation tools. *IOSR Journal of Electronics and Communication Engineering*, 5(2):54–60.

Chhimwal, P., Rai, D. S., and Rawat, D. (2013b). Comparison between different wireless sensor simulation tools. *IOSR Journal of Electronics and Communication Engineering*, 5(2):54–60.

Cleveland, M. A., Brunner, T. A., Gentile, N. A., and Keasler, J. A. (2013a). Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle monte carlo simulations. *Journal of Computational Physics*, 251:223–236.

Cleveland, M. A., Brunner, T. A., Gentile, N. A., and Keasler, J. A. (2013b). Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle monte carlo simulations. *Journal of Computational Physics*, 251:223–236.

Dalle, O. (2012a). On reproducibility and traceability of simulations. In *Winter Simulation Conference (WSC'12)*.

Dalle, O. (2012b). On reproducibility and traceability of simulations. In Laroque, C., Himmelspach, J., Pasupathy, R., Rose, O., and Uhrmacher, A. M., editors, *P*roceedings of the 2012 Winter Simulation Conference, page 244.

Distributed Computing Group, E. Z. (2015). Sinalgo: Simulator for network algorithms. `http://dcg.ethz.ch/projects/sinalgo`. Accessed March/2015.

Doerel, T. (2009). Simulation of wireless ad-hoc sensor networks with qualnet. *Technische Universitat Chemnitz.*

Fujimoto, R. M. (2000). *Parallel and distributed simulation systems*, volume 300. Wiley New York.

Gregory, J. (2017). *Virtual reality.* Cherry Lake.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533.

Hill, D. R. C. (2015). Parallel random numbers, simulation, and reproducible research. *Computing in Science and Engineering*, 17:66–71.

Johnson, D. B., Maltz, D. A., Broch, J., et al. (2001). Dsr: The dynamic source routing protocol for multi-hop wireless ad hoc networks. *Ad hoc networking*, 5:139–172.

Khan, M. Z., Askwith, B., Bouhafs, F., and Asim, M. (2011). Limitations of simulation tools for large-scale wireless sensor networks. In *Proceedings of the 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, WAINA '11, pages 820–825, Washington, DC, USA. IEEE Computer Society.

Lin, F.-T., Kuo, Y.-C., Hsieh, J.-C., Tsai, H.-Y., Liao, Y.-T., and Lee, H.-C. (2015). A self-powering wireless environment monitoring system using soil energy. *IEEE Sensors Journal*, 15(7):3751–3758.

Liu, J., Perrone, L. F., Nicol, D. M., Liljenstam, M., Elliott, C., and Pearson, D. (2001). Simulation modeling of large-scale ad-hoc sensor networks. In *European Simulation Interoperability Workshop*, volume 200.

Lu, W., Gong, Y., Liu, X., Wu, J., and Peng, H. (2017). Collaborative energy and information transfer in green wireless sensor networks for smart cities. *IEEE Transactions on Industrial Informatics*.

Maia, G., Aquino, A. L. L., Guidoni, D., and Loureiro, A. A. F. (2013). A multicast reprogramming protocol for wireless sensor networks based on small world concepts. *Journal of Parallel and Distributed Computing*, 73:1277–1291.

Minakov, I., Passerone, R., Rizzardi, A., and Sicari, S. (2016). A comparative study of recent wireless sensor network simulators. *ACM Trans. Sen. Netw.*, 12(3):20:1–20:39.

Muruganandam, M. K., Balamurugan, B., and Khara, S. (2018). Design of wireless sensor networks for iot application: A challenges and survey. *International Journal Of Engineering And Computer Science*, 7(03):23790–23795.

Okuda, Y., Bryson, E. O., DeMaria, S., Jacobson, L., Quinones, J., Shen, B., and Levine, A. I. (2009). The utility of simulation in medical education: what is the evidence? *Mount Sinai Journal of Medicine: A Journal of Translational and Personalized Medicine*, 76(4):330–343.

Ould-Ahmed-Vall, E., Riley, G. F., and Heck, B. S. (2007a). Large-scale sensor networks simulation with gtsnets. *Simulation*, 83(3):273–290.

Ould-Ahmed-Vall, E. M., Riley, G. F., and Heck, B. S. (2007b). Large-scale sensor networks simulation with GTSNetS. *Simulation*, 83(3):273–290.

Pacheco, P. S. (1997). *Parallel programming with MPI*. Morgan Kaufmann.

Perkins, C., Belding-Royer, E., and Das, S. (2003). Ad hoc on-demand distance vector (aodv) routing. Technical report.

Raković, P. and Lutovac, B. (2015). A cloud computing architecture with wireless body area network for professional athletes health monitoring in sports organizations—case study of montenegro. In *Embedded Computing (MECO), 2015 4th Mediterranean Conference on*, pages 387–390. IEEE.

Rashid, B. and Rehmani, M. H. (2016). Applications of wireless sensor networks for urban areas: A survey. *Journal of Network and Computer Applications*, 60(2016):192–219.

Ribeiro, D. H., Lima, J. C., Aquino, A. L., and Oliveira, R. A. (2012a). A parallel simulator for large scale wireless sensor networks. In *XIII Simpósio em Sistemas Computacionais (WSCAD-SSC)*, pages 01–05.

Ribeiro, D. H., Lima, J. C., Aquino, A. L., Viana, L. P., and Oliveira, R. A. (2012b). Jsensor: A parallel simulator for wireless sensor networks and distributed systems. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 604–605. IEEE.

Ribeiro, D. H., Lima, J. C., Aquino, A. L. L., Viana, L. P., and Oliveira, R. A. R. (2012c). JSensor: A parallel simulator for wireless sensor networks and distributed systems. In *International Conference on Parallel Processing Workshops (ICPPW'12)*.

Riley, G. and Park, A. (2004a). PDNS: Parallel and distributed NS (web site). Last access October 22, 2016.

Riley, G. and Park, A. (2004b). Pdns-parallel/distributed ns.

Riley, G. F. and Henderson, T. R. (2010). The ns-3 network simulator. In *Modeling and tools for network simulation*, pages 15–34. Springer.

Robey, R. W., Robey, J. M., and Aulwes, R. (2011a). In search of numerical consistency in parallel programming. *Parallel Computing*, 37:217–229.

Robey, R. W., Robey, J. M., and Aulwes, R. (2011b). In search of numerical consistency in parallel programming. *Parallel Computing*, 37(4):217–229.

Silva, M. L., Ribeiro, D. H., Lima, J. C., Aquino, A. L., and Oliveira, R. A. (2013). Jsensor: Um simulador paralelo para redes de sensores em larga escala. In *XXXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (Salão de ferramentas)*.

Sundani, H., Li, H., Devabhaktuni, V., Alam, M., and Bhattacharya, P. (2011a). Wireless sensor network simulators a survey and comparisons. *International Journal of Computer Networks*, 2(5):249–265.

Sundani, H., Li, H., Devabhaktuni, V., Alam, M., and Bhattacharya, P. (2011b). Wireless sensor network simulators a survey and comparisons. *International Journal of Computer Networks*, 2(5):249–265.

Sundresh, S., Kim, W., and Agha, G. (2004). Sens: A sensor, environment and network simulator. In *Proceedings of the 37th annual symposium on Simulation*, page 221. IEEE Computer Society.

Varga, A. and Hornig, R. (2008a). 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).

Varga, A. and Hornig, R. (2008b). 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).

Vieira, M. A. M., Coelho Jr, C. N., da Silva, D., and da Mata, J. M. (2003). Survey on wireless sensor network devices. In *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA'03. IEEE Conference*, volume 1, pages 537–544. IEEE.

Vodel, M., Sauppe, M., Caspar, M., and Hardt, W. (2008). Simanet–a large scalable, distributed simulation framework for ambient networks. *Journal of Communications*, 3(7):11–19.

Weingärtner, E., Vom Lehn, H., and Wehrle, K. (2009). A performance comparison of recent network simulators. In *Proceedings of the 2009 IEEE International Conference on Communications*, ICC'09, pages 1287–1291, Piscataway, NJ, USA. IEEE Press.

Xiong, H., Huang, Y., Barnes, L. E., and Gerber, M. S. (2016). Sensus: a cross-platform, general-purpose system for mobile crowdsensing in human-subject studies. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 415–426. ACM.

Xue, Y., Lee, H. S., Yang, M., Kumarawadu, P., Ghenniwa, H. H., and Shen, W. (2007). Performance evaluation of ns-2 simulator for wireless sensor networks. In *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pages 1372–1375. IEEE.

Yick, J., Mukherjee, B., and Ghosal, D. (2008). Wireless sensor network survey. *Computer networks*, 52(12):2292–2330.

Yu, F. and Jain, R. (2011). A survey of wireless sensor network simulation tools. *Washington University in St. Louis, Department of Science and Engineering.*

Zeng, X., Bagrodia, R., and Gerla, M. (1998a). Glomosim: A library for parallel simulation of large-scale wireless networks. *SIGSIM Simul. Dig.*, 28(1):154–161.

Zeng, X., Bagrodia, R., and Gerla, M. (1998b). GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation (PADS'98).*