SAPIENZA
UNIVERSITÀ DI ROMA

Techniques for Accurate and Scalable Simulation of Spiking Neural Networks using Speculative Discrete Event Simulation

Dipartimento di Ingegneria Informatica Automatica e Gestionale
Dottorato di Ricerca in Engineering in Computer Science (XXXVI cycle)

Adriano Pimpini
ID number 1645896

Advisor
Prof. Alessandro Pellegrini

Co-Advisor
Prof. Roberto Beraldi

Academic Year 2023/2024

Thesis not yet defended

This thesis has been reviewed by the following external reviewers:

Prof. Cristian Axenie
Dr.rer.nat. Philipp Andelfinger

Techniques for Accurate and Scalable Simulation of Spiking Neural Networks using Speculative Discrete Event Simulation
PhD Thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: pimpini@diag.uniroma1.it, adriano.pimpini@gmail.com

Abstract

In the era of ubiquitous Artificial Intelligence and power-hungry Neural Networks, the brain offers prime inspiration for a faster, greener, more efficient, and arguably more effective alternative: Spiking Neural Networks (SNNs).

This thesis explores simulating SNNs using Parallel Discrete Event Simulation (PDES) with Time Warp. We present the motivations for this approach, the challenges that using it poses, and illustrate our solutions in-depth from both a theoretical and technical point of view, with emphasis on the latter.

With the ability to execute SNN simulation on a PDES support, we show how this simulation method allows for achieving significantly higher simulation accuracy with respect to the traditional Time-Stepped approach. In our experimentation, the traditional approach was shown to suffer from substantial drift from the expected network activity due to the compounding effect of inaccuracies. Higher accuracy is crucial to properly simulate and thus study biological neural networks in silico, as well as simulate analogical neuromorphic chips, but we also show it plays a fundamental role when using SNNs for AI by replicating recognition experiments and achieving higher classification accuracy, all while using simpler network topologies, with lower energy consumption.

Finally, our experimentation also highlights the high scalability of our approach thanks to effective utilisation of both parallel and distributed computing.

Acknowledgements

I knew embarking on a Ph.D. was not for the faint of heart. What I did not know was how much of a community effort it would prove to be.

First, I would like to express my deepest gratitude to my advisor, Prof. Alessandro Pellegrini, who accompanied me starting from my Master's thesis, through these years of intense research, to this final dissertation, and always did so with the utmost availability, humanity, integrity, and professionalism. Along these lines I would like to also give my thanks to Prof. Bruno Ciciani, for being my Ph.D. advisor until retirement, but even more for being a most excellent professor before that, with an unmatched ability to convey his passion for complex things, which led me to take a look around academia. And I obviously cannot forget to thank Dr. Andrea Piccione, exquisite colleague and friend, for his playful and vigorous approach to life and to research. Without them, I would not have lasted one month doing what we did.

Then I would like to thank my parents for their continued, relentless, gratuitous support, for checking in on me when I studied in the night, for picking up after me when I was focused on a deadline, and running after me when I was around Europe. Making sure I did not have a care throughout the entirety of my academic path, and, more in general, for the entirety of my life. I thank my grandparents, which I am blessed to have, and to have had. Always my biggest fans, suffering with me in times of stress, rejoicing with me—often more(!) than me—for having reached this important accomplishment and for every single intermediate milestone leading up to it. Then I thank my aunts, uncles, cousins and more, for taking interest in this endeavor of mine, for supporting me and cheering for me. None of this would have been possible without your support.

Thirdly, I thank my many friends. Ex-schoolmates and colleagues and more, *from all walks of life. Every single one of them has played a vital role in shaping who I have become. The ones I meet often, and the ones I meet seldom, but not for this less important. My thanks for giving me careless moments in which to escape from my worries and challenges, so that I could later come back and face them head on. As well as for providing different, valuable points of view regarding the many aspects of life, and in general for making me part of their lives even when it was hard for me to be present, whether physically or mentally.*

Finally, I would like to thank Denise. For being there every single day, through thick and thin, cheering me on, driving me to achieve the best possible results. Thank you for being there, even when not even I was there for myself. Thanks for making me part of your life, and for being part of mine. Your love and comprehension are gifts I do treasure deeply. I will always strive to support you the way you supported me in this adventure, and I know you will in future ones. Together.

This achievement goes to all of you.

Adriano Pimpini

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For centuries, the brain has fascinated humans, who wondered about its purpose and, more recently, about its inner workings. Written records touching on the brain's structure and treatment of traumatic brain injury are found on the Edwin Smith Papyrus, bringing us as far back as the 17^{th} century BC, in ancient Egypt. However, it was not until the work of Alcmaeon of Croton in ancient Greece that the brain was first linked to higher intellectual activities [6].

Through continued effort across the centuries, this fundamental understanding continued to evolve, culminating in the groundbreaking discovery of the neuron as the basic unit of the nervous system in the late 19^{th} century, which led us to a time of deeper, although still incomplete, understanding of the brain.

At the same time, thinkers from every era have been captivated by other aspects of the human experience: the ability to have complex thoughts, problem-solving, and the very nature of consciousness itself. These have always been shrouded in enticing mystery. Philosophers long debated about the nature of the mind and its relationship to the body, while folklore and modern fiction often present stories of inanimate objects becoming sentient. In more modern times, we find attempts at recreating complex behaviour through automata, mechanical devices mimicking human or animal actions.

Nowadays, we know that the brain is the organ that controls the functions of the body and interprets the information from the outside world, allowing us to think and act in complex ways. Furthermore, modern technologies such as brain imaging allow us to take a peek into the brain and observe its activity in real time.

It is not surprising then, that the progresses on the path of brain discovery have inspired many of the advancements in the Artificial Intelligence (AI) field. Indeed, the most obvious path towards replicating (or rather, emulating) intelligence that is currently being explored is that of replicating the brain's inner workings, or at least its behaviour. For this reason, the concept of Artificial Neural Networks (ANNs) has become a hot topic in computer and data science, and the most well-known and practised approach to AI in the current day.

ANNs are function approximation tools loosely inspired by the brain's structure. From the brain, they borrow the idea of having a multitude of interconnected small and simple components—the neurons—, the interaction of which gives rise to complex behaviour. The applications achieved with this approach are countless, ranging from speech recognition and improved computer vision, to more recent advances in generative tasks, such as generative adversarial neural networks or diffusion models for images and videos [60, 24], and language processing through large language models [59]. However, training and using ANNs requires significant amounts of computational resources and, additionally, they often struggle to replicate the brain's flexibility and adaptability.

This inherent limitation of ANNs spurred deeper research in the direction of models that better follow the brain's biological functioning. Spiking Neural Networks (SNNs) represent in this sense a significant step forward, as strong candidates as a computational tool, to execute AI tasks with extreme energy efficiency compared to ANNs, but also—and possibly more notably—as a modelling tool to better understand the brain's nature, structure, and inner workings.

The interest in SNNs has increased in the last decade [23]. Their momentum is strongly related to their expressive capabilities that allow them to mimic biological neural networks closely. This characteristic makes SNNs a perfect tool for research activities in disparate fields, such as medicine, neuroscience, or psychology, and, again, provides a non-negligible potential in Artificial Intelligence. At the same time, their significantly-reduced energy requirements open the way for specialised hardware applications in the form of *neuromorphic chips* [26, 52, 56]. These chips are regarded as one of the essential future steps in computing, as they introduce a level of parallelism, with reduced energy demand, that does not exist in today's hardware, including GPUs, FPGAs, and most AI accelerators.

Neuromorphic systems, in general, have an increased value due to their capability to perform processing asynchronously. Indeed, they rely on event-driven processing models to tackle complex computing problems, in a way similar to the human brain, which uses only a subset of its neurons and synapses to carry out tasks at maximum efficiency.

SNNs encode data in a temporal domain known as the *spike train* [9]. Due to this behaviour that evolves with time, their output cannot be simply computed with a one-shot function, however complex, but instead they need to be simulated. The event-driven processing model of SNNs makes them a perfect match with Discrete Event Simulation (DES) techniques. Nevertheless, simulating SNNs is exceptionally computationally intensive due to their sheer size and scale. This is reflected in non-negligible running times, making it difficult to obtain relevant simulation results in reasonable time. As an example [45], simulating 250ms of activity for a network of 11,250 neurons/127 million synapses on the well-known NEST simulator [21] can take more than 30 seconds on a single CPU core. At the same time, the research community is striving to simulate networks of size comparable to the mammalian brain's, containing on the order of 10^7 to 10^{11} neurons, with thousands of synapses per neuron on average [2, 29, 30, 34], which are considered very large networks.

The computational demand of large-scale SNN simulation is addressed by the vast majority of existing simulators by employing parallel/distributed execution. This is done by exploiting multicore CPUs (also in distributed environments) or by relying on accelerators such as GPUs [5, 10, 14, 18, 32, 42, 45, 71] or FPGAs [13, 63, 70]. Nevertheless, we observed that the trouble at scaling up faced by most state-of-the-art SNN simulation methods may stem from their use of conservative time-stepped synchronisation. Indeed, popular simulators rely on the time-stepped simulation approach, and the simulators that employ Parallel Discrete Event Simulation (PDES), regardless of whether they use a conservative synchronisation scheme such as the YAWNS algorithm [46] (e.g. [15]), or not (e.g. [55]), rely on recurring “heartbeat” events scheduled at constant intervals to trigger neuronal dynamics computation, essentially making them into time-stepped simulations.

However, SNN models have, in general, a low rate of neuron activity at any given time, which is strictly connected to their brain-inspired nature, which uses only a subset of the neurons for a given task. This space/time partitioning of the activities makes it perfectly suitable for exploiting speculative simulation adhering to the Time Warp synchronisation protocol [33]. The hypothesis was that this choice could produce non-negligible simulation speedups, especially on large-scale computing infrastructures, as it has been shown that speculative PDES can be deployed on millions of (distributed) CPU cores [3].

At the same time, spikes carry a piece of information which is either binary (i.e., a neuron has spiked) or with a tiny payload (e.g., the intensity of the spike) depending on the nature of the model and its implementation—in general, the behaviour of spikes is somewhat homogeneous in a simulation. Similarly, the state of each neuron is reduced in size, and the time complexity of the execution of a single event can be significantly fine-grained. These aspects could make the employment of Time Warp-based PDES sub-optimal, as the housekeeping cost might not be paid off by forward-processing activities [20].

1.1 Thesis objectives and Contributions

In this work, we present modelling methodologies and PDES runtime-environment support for SNN models based on the ROOT-Sim Simulation Framework [51], which employs the Time Warp synchronisation protocol to enhance the scalability of simulations. This approach has a twofold goal. On the one hand, we tackle the complexity of deploying an SNN model on top of a speculative PDES runtime environment. The modeller should not need to worry about where the synapses are kept or how they are organised, nor should they care about the fact that spikes are, in fact, events that need to be sent (logically) one by one or, even worse, they should not need to schedule an event to check whether a spike should happen or not. Ideally, the modeller should be tasked with as few programming related actions as possible, made as simple as possible.

On the other hand, we focus on simulation accuracy. Indeed, many approaches relying on (time-stepped) synchronisation algorithms consider a fixed forward step in time (typically set on the order of tenths of milliseconds). In this way, the simulation results *approximate* the actual results. This is a well-known limitation [27] related to classes of neuron models with linear subthreshold dynamics, wherein some circumstances even some spikes can be missed. Our modelling methodology leverages the nature of discrete events together with an innovative numerical method and ad-hoc events management strategies, allowing us to obtain precise simulation results. Furthermore, we apply this approach to exponential synapses, which are much more complex than jump synapses typically dealt with in the literature that tackles spiking neural network simulation using DES.

The content of this thesis is partially based on the publications that appeared in international conferences, and are listed below at the label “SNN simulation”.

Furthermore, the author also contributed to the publications listed below at the label “Other Publications”, which, in varying degrees, have served as inspiration for the findings presented in this thesis.

Finally, as part of the commitment to the research community, the author has also been involved in the reproducibility initiative, for which the report listed below at the label “Reproducibility”, was produced.

SNN Simulation

[1] Adriano Pimpini. 2023. Towards accessible Parallel Discrete Event Simulation of Spiking Neural Networks. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Orlando, FL, USA) *(SIGSIM-PADS '23)*. Association for Computing Machinery, New York, NY, USA, 60–61.

[2] Adriano Pimpini, Andrea Piccione, Bruno Ciciani, and Alessandro Pellegrini. 2022. Speculative Distributed Simulation of Very Large Spiking Neural Networks. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Atlanta, GA, USA) *(SIGSIM-PADS '22)*. Association for Computing Machinery, New York, NY, USA, 93–104.

[3] Adriano Pimpini, Andrea Piccione, and Alessandro Pellegrini. 2022. On the Accuracy and Performance of Spiking Neural Network Simulations. In *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 96–103.

Other Publications

[1] Xiaorui Du, Andrea Piccione, Adriano Pimpini, Stefano Bortoli, Alois Knoll, and Alessandro Pellegrini. 2024. HUILLY: A Non-Blocking Ingestion Buffer for Timestepped Simulation Analytics. In *Proceedings of the 24th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE.

[2] Xiaorui Du, Andrea Piccione, Adriano Pimpini, Stefano Bortoli, Alessandro Pellegrini, and Alois Knoll. 2024. Online Analytics with Local Operator Rebinding for Simulation Data Stream Processing. In *2024 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (Urbino, Italy) *(To appear)*.

[3] Xiaorui Du, Adriano Pimpini, Andrea Piccione, Zhuoxiao Meng, Anibal Siguenza-Torres, Stefano Bortoli, Alois Knoll, and Alessandro Pellegrini. 2023. Autonomic Orchestration of in-situ and In-Transit Data Analytics for Simulation Studies. In *Proceedings of the Winter Simulation Conference* (San Antonio, Texas, USA) *(WSC '23)*. IEEE Press, 781–792.

[4] Adriano Pimpini and Alessandro Pellegrini. 2024. RBlockSim: Parallel and Distributed Simulation for Blockchain Benchmarking. (Submitted, awaiting review).

Reproducibility

[1] Adriano Pimpini. 2022. Reproducibility Report for the Paper: "Evaluating Performance of Spintronics-Based Spiking Neural Network Chips using Parallel Discrete Event Simulation". In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Atlanta, GA, USA) *(SIGSIM-PADS '22)*. Association for Computing Machinery, New York, NY, USA, 138–140.

1.1.1 Structure of this Thesis

This thesis is structured as follows. After introducing the work's background and motivations in the first and current Chapter, we move on to introduce Spiking Neural Networks in Chapter 2, and present the neuron model we use throughout this work, two synapse models and the simulation method classically used for SNN simulation: time-stepped simulation. Then, in Chapter 3, we present the approach we propose to use for SNN simulation, Parallel Discrete Event Simulation. We explain the basic concepts of PDES and explain how the previously presented neuron model can be adapted to be simulated using PDES. Furthermore, we introduce the runtime supports we implemented and used to achieve scalable and accurate simulations using PDES. Chapter 4 presents the Python interfaces implemented to allow for easier and faster development of SNN simulation models. Chapter 5 deals with the experimental evaluation of the approach, from a perspective of performance and accuracy. It also presents the work done to evaluate the capability of SNNs simulated using PDES to execute AI workloads and evaluate the performance of hardware deployments. Finally, Chapter 6 concludes the thesis, and proposes future research directions in line with the presented work.

1.2 Reference Implementations and Benchmarks

We now proceed to present the various models and benchmarks used in creating research for this thesis. All of the models and benchmarks have been implemented on top of runtime supports (see Section 3.5) developed for the ROOT-Sim Simulation Framework [51] specifically for the purpose of supporting this research. The source code can be found online[1]. Whenever possible, we opted-in to the Reproducibility of Computational Results supported by the ACM.

[1] `https://github.com/ROOT-Sim/core`

1.2.1 Simulation Frameworks

ROOT-Sim

ROOT-Sim[1] [51] is a general-purpose, parallel and distributed, high performance discrete event simulation framework, leveraging optimistic synchronisation using the Time Warp algorithm for speculative execution of events.

All the simulation runtime supports implemented for this research have been implemented on top of ROOT-Sim.

NEST

NEST[2] [22] is an open-source, time-stepped, spiking neural network simulator implemented in C++. Boasting a multitude of active contributors and almost 20 years of history, it comes prepacked with “over 50 neuron models many of which have been published” and “over 10 synapse models”. Users can implement new custom neuron and synapse models.

NEST can run parallel simulations through OpenMP, using the worker thread paradigm for parallel computing, while distributed simulations are supported via MPI. Inter-process communication and neuron distribution is handled transparently inside of NEST.

Brian

Brian [64] is an open-source time-stepped simulator written in Python. Its main focus is ease of use, but provides a series of interesting facilities that allow for great flexibility and high performance.

The simulation structure is simple and follows few key steps:

1. The runtime is initialised automatically upon importing Brian.
2. Neurons are declared and initialised through the NeuronGroup class.
3. Synaptic connection between two NeuronGroups is declared with a SynapseGroup object.

[2] `https://nest-simulator.org`

4. Synapses are created inside a SynapseGroup, using one of the provided connection methods.
5. Other inputs are created (e.g. input from a poissonian population of neurons).
6. A SpikeMonitor, or another similar object, is used to gather statistics about a target neuron, or (slice of) neuron group.
7. Created objects are added to the simulation.
8. The simulation is run for a selected amount of time.
9. Data is gathered, object states can be manually modified. These last two steps can be repeated ad libitum.

When declaring the neurons, the differential equations describing the state and its evolution have to be provided in **string format**. The equations are then parsed with SymPy, a library for symbolic mathematics, and is prepared to be solved with the preferred integration method at each time step. The need to provide differential equations is also found when creating synapses, both in modelling the synapse state, and the effect of spikes which must "be expressed as (possibly delayed) one-off changes".

Brian's internal code that executes models (i.e. that solves the model equations) is written and compiled on-the-fly when initialising the simulation. Based on the model, high-level code in string format is generated, which is then transposed into an intermediate representation that is optimised and then compiled into C++ code. At the time of experimentation, Brian lacked multi-threading support.

1.2.2 Synthetic Models

CUBA benchmark

The standard current-based (CUBA) synaptic interactions benchmark [9] is inspired by a study on signal propagation in leaky integrate-and-fire (LIF) models [69]. As the name suggests, this benchmark entails simulating a network of LIF neurons that communicate using current-based (CUBA) synaptic interactions. The network consists of two distinct populations of neurons: excitatory neurons, which make up the vast majority (80%) of the network, and inhibitory neurons, comprising the remaining 20%. All neurons are connected randomly to one another using a connection probability of 2%.

For the purpose of this research, the CUBA model proved useful for performance evaluation: owing to the parametric nature of the topology of its network, it provided an excellent infinitely-scaling stress-test. The network used for the majority of the experimentation we conducted comprises 300,000 neurons (240,000 excitatory and 60,000 inhibitory), leading to approximately 1.8 billion synapses.

2 neurons model

This network is composed of two neurons, the input neuron N_1 and the output neuron N_2. N_1 receives a constant external current of 1,800 mV, and its output is propagated to N_2 through an exponential synapse with a weight of 5,000. N_2 receives no other input. The output of N_2 is monitored, and its spikes are collected. The simplicity of this network allows us to easily evaluate simulation accuracy.

1000 neurons

This model is composed of a network of 1,000 LIF neurons. The neurons are divided into one input layer and three “passive” layers, each of which has two populations, one excitatory and one inhibitory. The input layer has 100 excitatory neurons, each of which receives a constant current input, and each of the three layers has 200 excitatory and 100 inhibitory neurons. It employs a connectivity map, reported in Table 5.3, to randomly connect the populations according to connection probabilities. This configuration aims to model a small network to statistically compare the observed spiking frequencies against those from state-of-the-art simulators.

Feed-forward Precision Benchmark

For the purpose of carrying out accuracy experiments, a synthetic network model consisting of 1,000 neurons was built. The network is acyclic, divided into four layers: Input, L1, L2, and Output. The network topology scheme is found in

Table 1.1. Connectivity map for the 1000 neurons synthetic benchmark.

		to					
		L1e	L1i	L2e	L2i	L3e	L3i
from	In	0.292	0.192	0.049	0.237	0.169	0.115
	L1e	0.224	0.293	0.106	0.254	0.438	0.099
	L1i	0.135	0.025	0.409	0.25	0.309	0.271
	L2e	0.165	0.177	0.122	0.032	0.491	0.3
	L2i	0.448	0.319	0.08	0.207	0.225	0.201
	L3e	0.395	0.123	0.265	0.215	0.476	0.174
	L3i	0.223	0.276	0.358	0.028	0.065	0.188

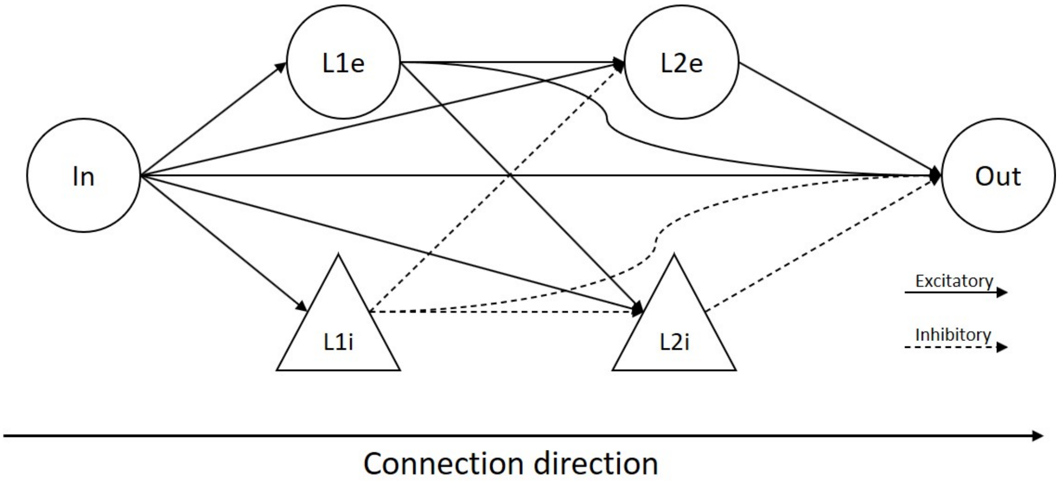


Figure 1.1. Schema of the Feed-forward Precision Benchmark.

Figure 1.1. The Input layer comprises 100 excitatory neurons, which receive a constant input current of $1800pA$. Layers L1 and L2 both comprise two populations of 100 excitatory (L1e/L2e) and 100 inhibitory neurons (L1i/L2i). The Output layer consists of 100 neurons. Synapses all have fixed weights of $200pA$ with a delay of $1.5ms$ when excitatory and a weight of $-600pA$ and delay of $0.8ms$ when inhibitory, as specified in Table 1.4. The neuron parameter specification is reported in Table 1.2.

This model is built to be comparable across different simulators. As such, it does not make use of randomness during the network simulation, so as to avoid differences stemming from (pseudo) random number generator implementations. Additionally, to render the network execution comparable across simulators, the network topology and relevant parameters (initial membrane potential, input current, synaptic weight,

Table 1.2. Parameter specification for the precision benchmark.

Populations and inputs

Name	Input	L1e	L1i	L2e	L2i	Output
Population size	100	200	200	200	200	100

Neuron Model

Name	Value	Description
τ_m	10 ms	Membrane time constant
τ_{ref}	2 ms	Absolute refractory period
τ_{syn}	0.5 ms	Postsynaptic current time constant
C_m	250 pF	Membrane capacity
V_{reset}	-65 mV	Reset potential
V_{th}	-50 mV	Fixed firing threshold

Table 1.3. Connectivity map for the Feed-forward Precision Benchmark.

		to					
		In	L1e	L1i	L2e	L2i	Out
from	In	-	0.292	0.192	0.049	0.237	0.169
	L1e	-	-	-	0.106	0.254	0.438
	L1i	-	-	-	0.409	0.250	0.309
	L2e	-	-	-	-	-	0.491
	L2i	-	-	-	-	-	0.225

Table 1.4. Synaptic parameter specification in the Feed-forward Precision Benchmark.

Name	Value	Description
w_{exc}	200 pA	Excitatory synaptic strength
w_{inh}	-600 pA	Inhibitory synaptic strength
d_e	1.5 ms	Excitatory synaptic transmission delay
d_i	0.8 ms	Inhibitory synaptic transmission delay

synaptic delay) are generated with a script into a configuration file, which then is loaded by the models of each simulator, leading to the exact same topology and initial conditions for every single neuron in all cases. Table 1.3 reports the connectivity map used when generating the topology.

1.2.3 Real-world Models

Local Cortical Microcircuit Model

As real-world model we used the local cortical microcircuit model developed by T. C. Potjans and M. Diesmann [57]. This model addressed a critical limitation in local cortical microcircuit models: while numerous such models existed, the simulated activity diverged from in-vivo recordings. While the prevailing approach focused on incorporating more complex neuron models, the researchers hypothesised that the discrepancy stemmed from inaccurate network connectivity.

They argued that existing connectivity maps, derived from anatomical and electrophysiological data, were insufficient. To address this, they developed a novel connectivity map that integrated insights from anatomy, electrophysiology, photostimulation, and electron microscopy studies. Their approach algorithmically combined the diverse datasets, accounting for the inherent biases of each experimental methodology.

The model comprises 4 layers of cortex, named 2/3, 4, 5, and 6, each with an excitatory and an inhibitory neuron population, for a total of 77,169 neurons. Layers are connected to one another according to probabilities present in a *connectivity map* reporting the 64 connection probabilities between the 8 populations. The synapses in the model are static. The number of neurons per population are chosen according to [8]. Furthermore, every layer can receive a background input in the form of a continuous current and input from an external thalamocortical neuron population. The thalamic neurons are implemented as Poisson neurons [53] with a fixed spiking rate.

Figure 1.2 shows the network representation as presented by the authors in [57]. Table 1.5 contains the neuron parameters: population sizes for each population, external inputs for e each population, and the neuron's physical properties. Table 1.6 contains the connectivity map and the synaptic connection parameters.

Table 1.5. Neurons and populations parameter specification.

Populations and inputs

Name	L2/3e	L2/3i	L4e	L4i	L5e	L5i	L6e	L6i	Th
Population size, N	20683	5834	21915	5479	4850	1065	14395	2948	902
External inputs, k_{ext}	1600	1500	2100	1900	2000	1900	2900	2100	n/a

Neuron Model

Name	Value	Description
τ_m	10 ms	Membrane time constant
τ_{ref}	2 ms	Absolute refractory period
τ_{syn}	0.5 ms	Postsynaptic current time constant
C_m	250 pF	Membrane capacity
V_{reset}	-65 mV	Reset potential
V_{th}	-50 mV	Fixed firing threshold
θ	15 Hz	Thalamic firing rate during input period

Table 1.6. Connectivity and Synaptic parameter specification.

Connectivity

		from								
		L2/3e	L2/3i	L4e	L4i	L5e	L5i	L6e	L6i	Th
to	L2/3e	0.101	0.169	0.044	0.082	0.032	0.0	0.008	0.0	0.0
	L2/3i	0.135	0.137	0.032	0.052	0.075	0.0	0.004	0.0	0.0
	L4e	0.008	0.006	0.050	0.135	0.007	0.0003	0.045	0.0	0.0983
	L4i	0.069	0.003	0.079	0.160	0.003	0.0	0.106	0.0	0.0619
	L5e	0.100	0.062	0.051	0.006	0.083	0.373	0.020	0.0	0.0
	L5i	0.055	0.027	0.026	0.002	0.060	0.316	0.009	0.0	0.0
	L6e	0.016	0.007	0.021	0.017	0.057	0.020	0.040	0.225	0.0512
	L6i	0.036	0.001	0.003	0.001	0.028	0.008	0.066	0.144	0.0196

Name	Value	Description
$w \pm \delta w$	87.8 ± 8.8 pA	Excitatory synaptic strengths
g	-4	Relative inhibitory synaptic strength
$d_e \pm \delta d_e$	1.5 ± 0.75 ms	Excitatory synaptic transmission delays
$d_i \pm \delta d_i$	0.8 ± 0.4 ms	Inhibitory synaptic transmission delays

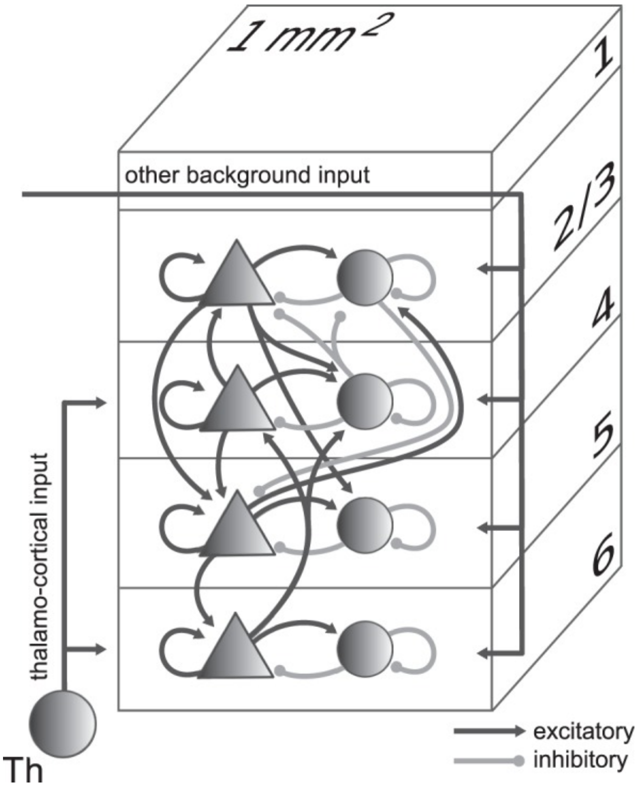


Figure 1.2. Potjans and Diesmann's local cortical microcircuit model structure. Excitatory populations are represented by triangles, inhibitory populations are circles.
Source: [57]

Chapter 2

Spiking Neural Networks

The ongoing pursuit of understanding and mimicking the brain's remarkable abilities to learn, generalise, and perform complex tasks drives research in Neural Networks, with the incredible efficiency of biological brains making them a natural source of inspiration. Various approaches have been developed to imitate the brain, each with its strengths and weaknesses, and different degrees of similarity with the original biological structure. Spiking Neural Networks take the concept further, striving for a higher degree of fidelity to the brain's structure and function. The intricate relationship between SNNs and biological neural information processing, requires a primer on our current knowledge of this aspect of the brain's inner workings.

2.1 From Biological Neurons to Spiking Neural Networks

The brain is a complex system, composed of a huge number of simple functional units: the neurons. A neuron (see Figure 2.1) consists of a cell body called *soma*, *dendrites*, and an *axon*. The axon and dendrites are filaments extruding from the soma which usually is, instead, compact. While the axon sparsely branches and can extend for surprising lengths (up to one meter in humans), dendrites do not travel far from the soma, but produce abundant branching. We can see the dendrites as the input channels of the neuron, while the axon is used for the output. At the tip of the axon's branches are axon terminals, where the neuron transmits signals across a *synapse* to another neuron's dendrite. In Neural Networks which strive for a higher degree of biological accuracy while maintaining performance, attention is currently placed on modelling of synapses and their strength: dendrites are abstracted away, and the axon's role is incorporated in other aspects of synapse modelling. Indeed, the axon's role can still be recognised in topological aspects, such as the neuron's increased probability of forming connections with neurons that are topologically closer, and the signal transmission delay. The latter depends both on the type of synapse and the point of the axon body at which the synapse lies.

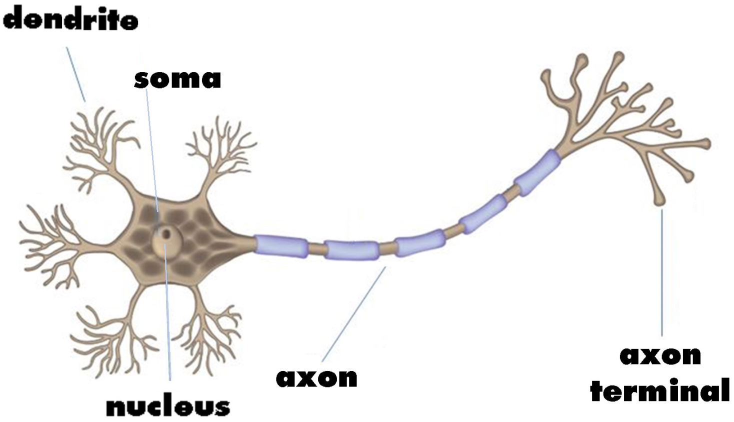


Figure 2.1. Representation of a neuron.
Source: "Neural Networks with R"

Neurons have plasma membranes with embedded voltage-gated ion channels. The membrane—among other things—electrically separates the inside of the cell with the outside, effectively creating what can be seen as a capacitor. The ion channels embedded in the membrane are sensitive to changes in its electric potential, which influences their opening and closing: the higher the potential is, the more these channels open, allowing more ions to flow through the otherwise ion-impermeable membrane. When the membrane potential is close to the *resting potential* the ion channels are completely closed. When the potential rises they open up, until it hits a precise *threshold voltage* for which a great number of (sodium) ions is allowed to flow inside the cell, starting an explosive chain reaction further raising the cell's membrane potential, causing more channels to open, and so on. The rapid rise of potential causes an inversion of the plasma membrane polarity, which rapidly deactivates the sodium ion channels, trapping the sodium (Na$^+$) ions inside the cell. The inversion of the membrane polarity is called *action potential* [4] (or signal, or **spike**) and propagates along the body of the cell, that is to say along the axon, to ultimately reach the synapses and propagate to the post-synaptic neurons. Note that the depolarisation is temporary, as the polarity inversion opens potassium (K$^-$) ion channels, which in turn let potassium ions flow outside of the membrane, returning the membrane potential to a negative value over a short period of time.

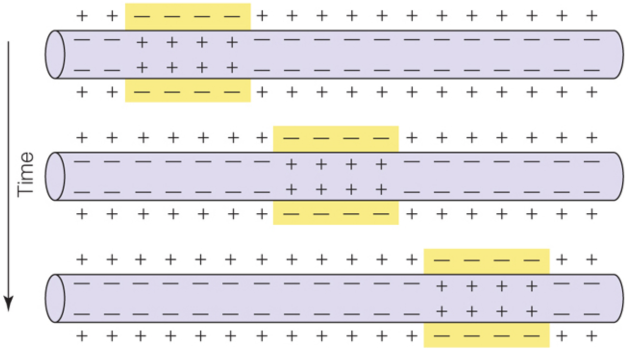


Figure 2.2. A visualisation of the action potential propagating through the axon with time.
Source: `https://www.macmillanhighered.com/BrainHoney/Resource/6716/digital_first_content/trunk/test/hillis2e/hillis2e_ch34_2.html`

After the depolarisation is reverted, the cell, that usually has potassium ions inside and sodium ions outside, is back to having negative electric potential, but with potassium ions outside and sodium ions inside the membrane. This situation is reverted by the sodium potassium pump, which restores the initial conditions by actively transporting the sodium outside of the membrane and the potassium back inside, at the same time. Until this process is completed, the membrane potential cannot rise, as such the time interval between the generation of the action potential and the completion of the “resetting” of the potential is called *refractory period* of the neuron. During the refractory period, the neuron is essentially unaffected by further incoming spikes.

Building on top of this knowledge, SNNs, differently from other types of Artificial Neural Network (ANNs) utilise a unique class of neuron: the spiking neuron. These neurons rely on communication through electrical signals (spikes) transmitted via the synapses connecting them.

Unlike their counterparts in ANNs, spiking neurons are stateful, meaning their internal state influences their behaviour. Synapses may also exhibit stateful properties. The key difference in information processing lies in how these neurons generate outputs. While neurons in other ANNs produce and propagate an output whenever they receive an input, spiking neurons only propagate information when a specific condition is met: much like biological neurons, spiking neurons only fire (generate a spike) when their membrane potential reaches a specific threshold value.

When a spiking neuron fires, it generates a spike that is propagated to the neurons it is connected to. The receiving neurons then react by increasing or decreasing their membrane potential accordingly, over time. To reach another neuron, the spike passes through the synapse connecting the two. The synapse is weighted and introduces a **transmission delay**. Indeed, a fundamental aspect that differentiates SNNs from other ANNs is the role that **time** plays: while in other classes of ANNs the computation and propagation of the output is, in general, instantaneous, spiking neurons need to wait for their membrane to charge over time. When the threshold potential value is reached, they fire and, after a transmission delay, the post-synaptic neurons (i.e. those connected on the receiving end of the synapses connected to the spiking neuron's axon) receive the signal. As such, information is not only encoded in the way synaptic weights change the amplitude of spikes, but in their timing as well.

While the reader may recognise a time-dependent behaviour similar to that taking place in Recurrent Neural Networks (RNNs), it is worth emphasising how SNNs' time dependence differs from that of RNNs: the former evolve through time by having their state be influenced by past events, while the latter do so by feeding their output back to themselves or to special memory units to use in the next computational step.

Spiking neurons models are derived from experimental observation of biological neurons' behaviour. Starting from the emergent behaviour of the neuron, electronic circuits that approximate it are devised. The structure and parameters of the circuits are derived by feeding the neuron with different input currents and seeing what the response to the various different stimuli is. Thanks to the knowledge of a neuron's anatomy, the observations that follow can be made:

- The isolating properties of a neuron's plasma membrane give rise to the membrane capacitance C_m.
- The potential between the two sides of the membrane (that we refer to as membrane potential V_m) is what kick-starts the action potential propagation once it reaches a target threshold value V_{th}.
- In the absence of incoming stimuli, the membrane potential gradually resets to a resting value V_r; this also holds true after the action potential is generated and the sodium potassium pump is done reverting the neuron back to its resting state, that is, after the refractory period τ_{ref} is elapsed.
- For the membrane potential to rise, there has to be some kind input current I, which is the sum of the stimuli coming from pre-synaptic neurons. Additionally, an external current I_{ext} can be supplied (e.g. for experimental observation).

These observations provide a solid foundation of the mathematical modelling process for a biological neural model. At the same time, from the presence of the capacitance follows that a spiking neuron is stateful (the minimum state being just the membrane potential at a given time) and that such state evolves with time.

Since this membrane potential evolves with incoming spikes, networks of spiking neurons require simulation through time to capture their behaviour. This time-dependent nature of SNNs makes them computationally expensive to execute, which is a defining characteristic and a key challenge explored throughout this work.

The computationally expensive nature of SNN simulation could raise doubts about the necessity and usefulness of SNN simulation. However, continued research in this area is motivated by several potential benefits.

The first reason motivating SNN research lies in its potential to simulate, with increasing efficiency and precision, specific aspects of the human brain's behaviour, while aiming to eventually simulate the entirety of the brain itself. This capability would be invaluable in various fields. In neuroscientific research, SNNs could provide a powerful tool for studying complex cognitive functions like learning and decision-making. Similarly, the field of medicine could benefit from the ability to simulate brain diseases, their evolution, and their response to potential treatments using SNNs.

The second reason stems from the fact that spiking neurons are modelled as electronic circuits: they can easily be implemented in hardware. A series of *neuromorphic chips* have already been created and commercialised (see IBM's TrueNorth neuromorphic processor [12]). This removes the cost associated with simulation, and a series of advantages arise with respect to all other ANNs:

- No approximation: since the electronic components are physically present, there is no approximation stemming from the precision limit that is inherent of computer simulations.
- Computation is inherently and naturally parallel: what actually happens in a chip with physically implemented neurons is essentially signal processing. No orchestration or communication between worker threads (which then may or may not share memory, etc.) is needed.
- Locally stored state: state is stored in the components, which means no moving data back-and-forth from memory to CPU and vice-versa, which is a crippling bottleneck when running networks on Von-Neumann machines.
- Energy and power efficiency: specialised circuitry is vastly more energy and power efficient than general purpose computational units, whether it be CPUs or GPUs we compare it with (see, e.g. [41]).

However, such huge advantages come with the drawback of high hardware design and manufacturing cost. This is also because SNNs can be huge, requiring a great number of neurons and an even greater number of synapses. Before investing in hardware production or acquisition, conducting thorough research is crucial. One may want to compare firing rates and general network behaviour with those of the actual natural neural network they are trying to replicate, or any other correctness metric of interest. This is especially important when no access to a neuromorphic chip is available, or when implementing a kind of neuron or synapse that existing neuromorphic chips might not be able to properly replicate.

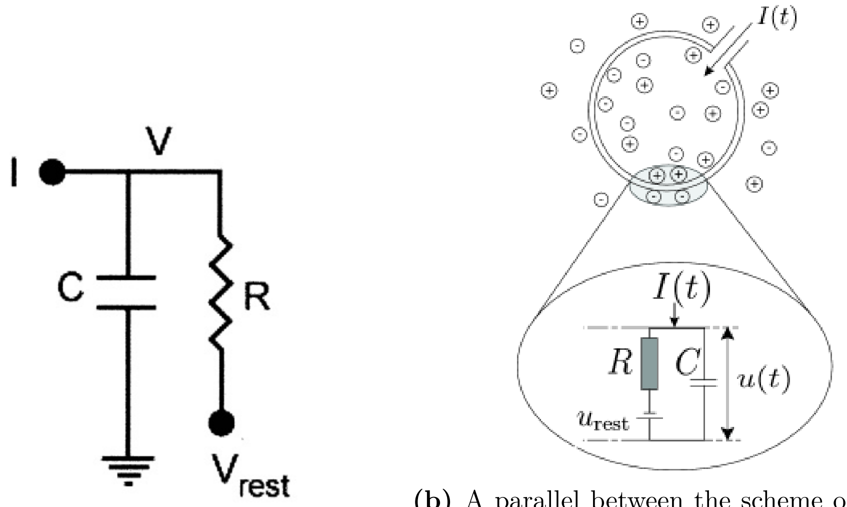
SNN simulation then serves as a cost-effective and time-efficient method for prototyping new hardware solutions and validating new SNN approaches, especially when dedicated hardware is unavailable or shows physical limitations. Furthermore, if between now and the release of neuromorphic chips to consumer market (and their widespread adoption as hardware accelerators) a point is reached in which SNN simulation becomes very efficient, we could be able to exploit SNNs to perform tasks without relying on specialised hardware accelerators.

2.2 The Leaky Integrate and Fire Spiking Neuron

The Leaky Integrate and Fire (LIF) neuron model is one of the simplest, yet powerful, models for spiking neurons. First developed by L. Lapicque in 1907 [1], it models the neuron as a leaky integrator. This simple model offers a lightweight representation of a neuron, from the standpoints of both computational and memory costs. Furthermore the model is analytically tractable. These advantages have made the LIF neurons into a popular choice for building large-scale SNNs, for applications in deep learning and artificial intelligence, as well as for biological modelling.

The LIF model (in Figure 2.3) comprises all the fundamental aspects we have mentioned in Section 2.1. In Figure 2.3a, we find the circuit devised by L. Lapicque, in which we find the components to model the membrane capacitance C, the membrane potential $V(t)$, a resting potential V_{rest}, an input current $I(t)$. The model also has a resistance R in parallel with the capacitor, which allows us to compute the time constant of the "leaky integrator" $\tau_m = RC$. The threshold potential V_{th} is not presented in the scheme, but it was postulated by Lapicque that "when the membrane capacitor was charged to certain threshold potential, an action potential would be generated and the capacitor would discharge, resetting the membrane potential" ([1]). The only aspect missing from the original model that we instead consider, thanks to improved knowledge of neuronal dynamics, is the presence of the refractory period, τ_{ref}.

To derive the model's equation, we use Figure 2.3a as a reference. First of all,



(a) Lapicque's LIF model as a circuit.
Source: [1]

(b) A parallel between the scheme of a neuron and the LIF model circuit.
Source: `https://neuronaldynamics.epfl.ch/online/Ch1.S3.html`

Figure 2.3

because of the law of current conservation, the input current $I(t)$ is split into two components: I_R and I_C, respectively passing through the resistor and charging the capacitor.

$$I(t) = I_R + I_C \tag{2.1}$$

From Ohm's law we have for the resistive current:

$$I_R = \frac{V_R}{R} \quad \text{where} \quad V_R = V(t) - V_{rest} \tag{2.2}$$

with V_R the voltage between the extremes of the resistor. For I_C, we have:

$$I_C = C\frac{\mathrm{d}V}{\mathrm{d}t} \quad \text{because} \quad C := \frac{q}{V} \quad \text{and} \quad I_C := \frac{\mathrm{d}q}{\mathrm{d}t} \tag{2.3}$$

From Equations 2.2 and 2.3 we get:

$$I(t) = \frac{V(t) - V_{rest}}{R} + C\frac{\mathrm{d}V}{\mathrm{d}t} \tag{2.4}$$

By multiplying Equation 2.4 by R we obtain the standard form:

$$\tau_m \frac{\mathrm{d}V}{\mathrm{d}t} = -(V(t) - V_{rest}) + RI(t) \tag{2.5}$$

Where $\tau_m = RC$ is the membrane time constant of the neuron.

In Equation 2.5 is the differential equation modelling the neuronal behaviour. We again notice how the firing threshold V_{th} does not figure in the equation, but is to be taken into consideration when using it. Furthermore the behaviour of the neuron after the membrane potential hits the threshold potential—namely the sudden rise of the membrane potential until the polarity is reversed, followed by an abrupt drop in $V(t)$, and the start of a refractory period—is not specified; we now know however that such behaviour is marginal for this model and can be approximated by resetting V to V_{rest} without any noticeable consequences. This holds true unless someone were to be trying to also keep into account the inductive currents that the abrupt rise of V would cause in neighbouring neurons (that some have theorised could play a role in neural ensemble behaviour), but one could argue that to explore and study such a complex process some other, more accurate, neuron model should be used rather than the LIF.

To include in the mathematical model the spiking non-linear behaviour once the potential reaches the threshold value V_{th}, as well as the refractory period, we can expand Equation 2.5 with Equation 2.6.

$$V(t) = V_{th} \wedge \hat{t} \in (t, t + \tau_{ref}] \implies V(\hat{t}) = V_{rest} \tag{2.6}$$

Despite its simplicity as an approximation of a real neuron, Lapicque's work has been a pioneering effort in the field, the effects of which persist to this day. Nowadays, extensions of the LIF neuron are found everywhere and still make up the majority of neurons used in simulations.

The Leaky Integrate and Fire neuron with current based exponential synaptic interactions is the reference neuron model used throughout this work.

2.3 Synapse Models

SNNs rely on the concept of synapse to model the intricate communication patterns between neurons. Synapses are not merely passive connections, but rather dynamic elements that can influence the flow of information. To capture this complexity, researchers have developed various synapse models, each offering different levels of biological realism and computational efficiency. In this section, two of the most commonly used synapse models—delta synapses and exponential synapses—are introduced.

2.3.1 Dirac-delta Synapses

The Dirac-delta synapse (or, simply, delta synapse, or jump synapse) is an idealised and simplified model of the synapse. When using this delta synapses, postsynaptic currents are delivered instantaneously and atomically upon receiving a spike, in a Dirac-delta fashion. This results in an instantaneous "jump" in the membrane potential of the receiving neuron.

Jump synapses are the most widespread synapse model for simulating SNNs using discrete event simulation: the simplicity and atomicity of the state update make it the obvious choice for ease of implementation. When the spike is delivered, the membrane potential is updated: if it surpasses the spiking threshold, the neuron spikes, otherwise it does not. This ensures no calculations need to be carried out to discover potential future spike times, resulting from the gradual application of the incoming spike.

While computationally extremely convenient, using delta synapses greatly reduces the expression potential of SNNs, which is tightly coupled to timing, both in spike delivery and application. Squashing down the synaptic signal dynamics, results in a fundamentally unrealistic behaviour.

2.3.2 Exponential Synapses

Instantaneous rise, exponential decay synapses (or more commonly exponential synapses) are a more complex synapse model, which models the postsynaptic current generated by a spike as an exponentially decaying function over time. While the incoming current is still modelled as instantaneously rising, it is applied over time to the membrane potential, which provides a more realistic representation of biological synapses when compared to delta synapses.

Equation 2.7 reports the differential equation describing the evolution of the input current the neuron receives when using this kind of synapse. Equation 2.8 instead reports the closed form equation, which we used in our modelling (see Section 3.4).

$$\frac{\mathrm{d}I(t)}{\mathrm{d}t} = -\frac{I(t)}{\tau_{syn}} \tag{2.7}$$

$$I(t) = e^{-\frac{\Delta t}{\tau_{syn}}} I_{spk} \tag{2.8}$$

The continuous nature of the postsynaptic current generated by exponential synapses—as opposed to the instantaneous nature of delta synapses—significantly increases the computational effort required to simulate them, regardless of the chosen approach (time-stepped or discrete event simulation). The challenges it poses for DES are explored in Section 3.4.

2.4 Classical Simulation Methods

The problem of accuracy and performance in SNN simulations is well-known in the literature [27]. In particular, it directly derives from the availability of multiple methods to solve the neuron model equations and handle spike events [25, 40, 36, 28]. The technique picked for a particular simulation directly determines the simulation speed and accuracy of the results [43].

To improve accuracy, several works [27, 72, 39, 48] have employed different numerical approaches or parameter estimation. Overall, resorting to a different numerical approach still depends on the underlying scheduling/synchronisation algorithm. In this sense, the work in [28] tackles the estimation of the error introduced by an SNN simulation. In particular, the authors show that it is possible to separate the numerical integration error from the spike-detection timing error, proper of time stepped simulations.

On the performance side, several works have explored the possibility of running SNN models on various hardware instances, such as GPUs [7, 68, 72], FPGAs [67, 37], or even heterogeneous systems [45]. Again, these works mainly focus on the deployment of time-stepped simulations.

The literature has also considered exploiting different discrete event simulation algorithms. In [55], the authors have shown that relying on speculative PDES simulation using the Time Warp synchronisation protocol can lead to non-minimal performance improvement when focusing on the TrueNorth Leaky Integrate and Fire (TNLIF) [12] architecture. However, their evaluation is limited to the TNLIF implementation provided, which uses delta synapses (see Section 2.3.1), and employs a "heartbeat" event, that is scheduled at constant intervals towards each neuron in order to carry out the computation of neuronal dynamics. In [65], discrete event simulation is used to provide better performance in the case of synapse models showing a spike latency, i.e. a delay exhibited in response to depolarisation. The research we presented in [53] is inheriting the usage of Time Warp synchronisation from [55] and then goes on to show that with proper event management strategies, it is possible to obtain good scalability even when using more complex instantaneous raise/exponential decay synapses, all while avoiding the use of heartbeat events.

This section illustrates the inner workings of the first approach to SNN simulation we consider in this work: Time-Stepped Simulation. Time-Stepped Simulation is the classical, most widespread approach for this matter. Later in this work, Chapter 3, explores in detail the approach we propose to achieve scalable and more accurate SNN simulation: Parallel Discrete Event Simulation using Time Warp.

2.4.1 Time-Stepped Simulations

As previously stated, the largest part of SNN simulations as, e.g., implemented by NEST [21] and Brian [64] simulators, rely on time-stepped algorithms. An example high-level pseudo-code implementing time-stepped simulation of SNNs is provided in Algorithm 1.

This kind of simulation approach is rather straightforward. In it, all neuron

Algorithm 1: Time-Stepped Simulation Algorithm.

```
$t = 0$
**while** $t < t_{end}$ **do**
    **foreach** *neuron* **do**
        process incoming spikes
        advance neuron dynamics by $\mathrm{d}t$
    **foreach** *neuron* **do**
        **if** $V(t) > V_{th}$ **then**
            reset neuron membrane
            **foreach** *connection* **do**
                send spike
    $t \leftarrow t + \mathrm{d}t$
```

states are updated periodically by evaluating an update function, and by processing any incoming spikes for the time-step. These spikes increase membrane potential, which is again evaluated numerically after every time interval $\mathrm{d}t$. After updating all neurons' states, the simulation algorithm checks which of them (if any) have a membrane potential V_m that has reached the spiking threshold, V_{th}. If this is the case for any neuron, spikes are sent from each of the ready-to-spike neurons to the respective postsynaptic neurons. To account for synapse delays, the typical strategy is to rely on some sort of future event queue, typically implemented as a circular array [9], that allows to keep track of what spike should be delivered to what neuron at what time(step) in the future.

The time complexity of this simulation algorithm can easily be calculated. The first inner loop accounts for neuron state updates. If there are n neurons in the network, the loop has an $O(n)$ cost. Considering that the physical time of the simulation is divided into intervals of the same size, the cost is $O(n/\,\mathrm{d}t)$ per unit of physical time. Concerning the second inner loop, if we call f the average firing rate of neurons per physical-time unit, assuming that on average each neuron is connected to s other neurons, the cost is $O(fns)$. Under general assumptions, it cannot be stated which of the two components impacts the overall cost more. We can therefore conclude that the cost of this algorithm *per physical-time unit* is:

$$O\left(\frac{n}{\mathrm{d}t} + fns\right) \tag{2.9}$$

In performing this calculation, we have assumed that activities related to the computation of neuron dynamics and spike delivery are negligible, although depending on the specific used neuron model and the complexity of the topology, it might not always be the case. Regardless, Equation (2.9) indicates that the overall computational cost of the time-stepped simulation grows with the network's size and the simulation's precision, which is one of the key points we explore experimentally in this work.

An additional issue with the time-stepped simulation algorithm is that spike timings are aligned to a grid defined by the time steps. Therefore, the final result approximates the actual behaviour of the network, even when the numerical methods used to compute differential equations provide exact results. Similarly, since the check on the threshold is carried out only at the time steps (see line 7 in Algorithm 1), some spikes may be missed. This is the second key point that we assess experimentally in this paper. We further explore the effects of snapping the spike timing and the threshold checking to the time step grid in Section 5.3.

2.5 Scaling SNN Simulations and Other Notable Simulators

Several works have proposed techniques to speed up or scale out SNN simulations. Notably, in [55], the authors have shown that relying on speculative PDES simulation using the Time Warp synchronisation protocol can lead to non-minimal performance improvement, especially in scenarios where only a subset of neurons, in a specific time window, are actively sending spikes to each other. While that work mainly focuses on the TrueNorth Leaky Integrate and Fire (TNLIF) [12] architecture, the seminal results in the paper can be deemed general.

In [45] the authors show the viability of running SNN simulations on top of heterogeneous hardware, possibly composed of CPUs, GPUs and FPGAs. The main contribution in the work is identifying portions of code in an SNN model that is repeatedly used in multiple simulations and/or runtime environments. This code is manually ported to the different hardware architectures. Conversely, the code specifically bound to the model is automatically transformed towards the target architecture.

The relevance of large-scale SNNs simulation is also reflected in the availability of multiple tools to design experiments and obtain simulation data. Several SNN simulators have been proposed in the literature, frequently focusing on scalability to large networks.

Among them are the simulators NEST [21] and Brian [64], that we presented in Section 1.2.1. Another notable simulator to mention is Neuron [11], which mainly focuses on biologically-accurate simulation. Provides facilities to observe internal aspects of neurons and synapses, such as the model's electrical, chemical, and topological evolution. The simulator can describe ion concentrations and the functioning of ion gates, the dynamics of ion diffusion, and more.

Other established simulators, such as CARLsim [14], NCS [32], NeMo [18], Nengo [5], HRLSim [42], and PCSIM [49] support multi-threaded execution on CPUs, some of them with support for execution on multiple nodes. GeNN [71] can be executed on a single GPU. Some simulators additionally support the execution in GPU clusters [14, 18, 32, 42, 71]. CARLsim [14] and NCS [32] support a CPU-GPU co-execution.

Chapter 3

Scalable Accurate Simulations

An SNN simulation can be seen as a collection of simulations of individual neurons that interact via the exchange of spikes. The changes in neuron state may trigger the emission of a spike, which is then delivered to the connected neurons. Since the spikes must be considered in the target neurons' future state updates, a neuron's state can only be consistently updated once it has received all spikes with smaller timestamps.

The computation of neuron state updates and the delivery of the generated spikes are the two principal operations implemented by SNN simulation platforms. Since most of them employ a time-step-driven approach, commonly used neuron models have been picked favouring those handily computable via some iterative numerical procedure, such as the Euler method.

Anyhow, the conservative synchronisation scheme typically employed by SNN simulators cannot efficiently deal with arbitrarily-low synaptic delays, which would otherwise force a costly synchronisation and spike delivery operation after each time-step. For this reason, many models include a minimum fixed delay in spikes transmission to reduce the necessary synchronisation steps and improve spike delivery performance.

To the best of our knowledge, prior to our research contributions listed in Section 1.1, the only SNN synapse model used in (P)DES were jump (a.k.a. delta) synapses (see Section 2.3.1). When a jump synapse transmits a spike, it is instantaneously applied to the membrane in a Dirac-delta fashion, resulting in a jump in the membrane potential. This behaviour is highly convenient for PDES simulations as it requires no further computation and perfectly fits the DES paradigm: if a spike results in the membrane potential surpassing the firing threshold, the neuron spikes, otherwise it does not.

In this work, we transition to the optimistic PDES paradigm *instantaneous raise/exponential decay synapses*, commonly called just exponential synapses (see Section 2.3.2). This type of synapse does not directly act on the membrane potential, but rather, it generates an instantaneous increase in the incoming current the neuron receives. The current's effects are applied over time to the membrane potential: the latter rises over time, charging similarly to an electronic capacitor. At the same time, the current's intensity decreases exponentially with time. This means the neuron might spike in the future, the hypothetical spike time (if any) has to be computed via numerical methods, and the resulting event enqueued.

In the rest of this chapter, we introduce Discrete Event Simulation, then outline the steps needed to represent this spiking model in a PDES runtime successfully, and finally we present the extensions to the traditional PDES facilities required to speed up the spike scheduling and delivery for SNNs.

3.1 Virtual Time

To talk about Discrete Event Simulation, the concept of Virtual Time (VT) [33] has to be introduced. As opposed to Wall Clock Time (WCT), which is the time that passes in real life, that a clock on the wall would measure, the "Virtual Time paradigm is a method of organising distributed systems by imposing on them a temporal coordinate system more computationally meaningful than real time" for the given application, e.g. when simulating a producer-consumer scenario, the concept of time could be based on the number of steps needed to produce and process units.

Using VT frees us from the need to evolve at the same speed at which wall clock time evolves, or to use a notion of advancement based on time altogether. As such, depending on the simulation complexity, needs, and the specific temporal coordinate system used, VT can pass much faster or much slower than wall clock time.

In the specific case of spiking neural networks, virtual time is related to wall clock time in its representation and meaning (one can simulate SNNs using seconds or milliseconds for VT).

3.2 Discrete Event Simulation

Discrete Event Simulation (DES) is a simulation technique that focuses on representing a system's behaviour through discrete events, which are significant moments that occur at specific points in simulated time. These events are considered instantaneous within the simulation's timescale. Phenomena that have a longer duration and cannot be considered instantaneous can be modelled by capturing their key moments. For instance, a quick computation in a client-server system might be a single event, while a longer process like database access or a download/upload could be modelled with two events: start and finish.

In DES the simulation is optionally partitioned into a series of Logical Processes (LPs), each of which represents a sub-portion of the simulation and receives events in the form of messages, responds to them by properly handling incoming events and consequently updating its state, and possibly scheduling new ones for the future. For the sake of simplicity, in this work we always assume the simulation uses logical processes.

For LPs, the passage of time happens with simulation time leaping from one event to the next: the state of the simulation is considered static, or rather in a state of inertia—meaning that its evolution follows a known trajectory—in between events. As such, DES is ideal to be used in systems whose evolution can easily be described by only taking into account some key points in time. On the other hand, modelling continuous systems with DES can be complex, as it might require closely spaced events or calculations about the future state in order to simulate continuous behaviour.

In this section, a vision of DES is presented in-depth through a systemic approach, followed by Parallel DES and the necessary support to execute discrete event simulation in parallel.

3.2.1 Systemic Approach to DES

We will now introduce Discrete Event Simulation using a systemic approach.

When using DES, the model is defined as the union of:

- the simulation states of Logical Processes, that when taken together describe and keep track of the system's evolution.
- a set E of events, which describe key phenomena happening during the evolution of the system and that cause the state to evolve in response to them.
- a transition function $\sigma(s, e) : S \times E \to S$ which describes the transition between two simulation states s and s' when an event $e \in E$ is received and handled (we can also refer to this as "execution" of an event).

Each event e has a timestamp T_e that corresponds to the Virtual Time (VT) at which it takes place.

Borrowing from event-driven programming, a DES model can be seen as "a set of event handlers, which capture the events generated by the same application (i.e., the simulation model) and, depending on the nature of the event, produce a state variation." (from [50]).

While handling an event e, a series of new events can be generated. As such, the notion of *causal dependency* between events is introduced: if an event e' is generated while processing event e, we say that e' *causally depends* from e. In other words, e' only exists as a consequence of the execution of event e. The causal dependence from e also holds for all the events that causally depend from e', since causal dependence is a transitive property. Due to the non-decreasing nature of (virtual) time, it has to hold that for all events e' causally dependent from e we have $T_{e'} \geq T_e$, as events happening in the (simulation) present or future may not influence the past. The notion of causal dependence might seem similar to the "happened-before" relationship in distributed systems [35]. Indeed, both worlds face challenges related to coordinating events across multiple entities, and ensuring a consistent global state. This shared challenge underscores the importance of enforcing causal dependence, which is a critical precondition to guarantee correctness of execution, especially in Parallel DES.

The simulation can be split into two components: the model and the kernel. While the model describes the system that we want to simulate with its event handlers, the simulation kernel is responsible of providing all the facilities that make this possible. This is typically achieved through the usage of APIs: the kernel "serves" the model's requests for whatever simulation related need it might have, and guides the execution from start to finish by running the *simulation loop*. As such, when a model generates a new event e' during the execution of event e, the event e' is passed along with its timestamp T_e to the kernel by using the exposed APIs, and the kernel will take care of enqueueing the event and raising it at the correct timestamp by waking the receiving LP. If a collision were to arise with respect to timestamps (i.e. when $T_{e'} = T_{e''}$ and $e' \neq e''$) the kernel resorts to tie-breaking policies to decide which of the two events is to be handled first. These policies have a very delicate job, as malformed tie-breaking could lead to biased or erroneous simulation.

3.2.2 Components of DES

To fulfil its simulation management duties, the simulation kernel relies on a series of concepts and components that will now be explored.

Simulation state. The state of the simulation is composed of the states of the various LPs into which the model is split. The evolution of these states depends on the transitions produced by the transition function $\sigma(s, e) : S \times E \to S$. In practice, the transition from one state to the other is operated by the event handlers.

Events. Events are the core concept of DES. Each simulation model defines a-priori the set E of event types that may take place at run time. This definition process also entails defining an appropriate programmatic event handler for every event $e \in E$. The simulation framework can define additional control events for the model to handle. An example of one such event in ROOT-Sim is the INIT event. It is scheduled by the kernel before starting the simulation, has timestamp 0 and serves the purpose of letting the model initialise all the needed variables and data structures representing the state to their initial values and schedule the first batch of events. While framework generated, a handler must be defined in the model for events of this kind, as the handling is anyway model-dependent.

Clock. The simulation kernel is also responsible of keeping track of the Virtual Time of the simulation. As we know, in DES VT does not move continuously, but rather it leaps from one event's timestamp to the next event's timestamp. When the execution is serial the clock can be a global variable as events are raised and processed in causal order, and no important distinction needs to be done between the time of the whole simulation, and that of the LPs. This changes for parallel execution, which will be discussed later.

Event Queue. The kernel is tasked with keeping track of events and delivering them at appropriate times. Event management is achieved through the use of the Event Queue (EQ). The event queue is a priority queue that holds events scheduled for the future: when the model wants to schedule a new event, it populates the event and calls the appropriate kernel API, which puts the event into the queue. When the handling of an event concludes (i.e. the handler function returns control to the kernel), the kernel goes on to extract the next upcoming event from the EQ. While being used in a write-heavy fashion, the actual implementation of the priority queue is irrelevant, as long as its performance is adequate with respect to the application needs. Given T_{min} the minimum timestamp of any event in the queue, no event with timestamp $T_{e_i} > T_{min}$ shall be raised by the kernel, as this would cause a causality violation, resulting in an incorrect execution. In non-parallel DES, raising events in order is simple.

Ending Condition. Simulation models often describe phenomena that do not stop on their own, but rather continue on evolving. In practical terms, this means that new events are continuously generated by the model, and the event queue never empties. Furthermore, there are phenomena for which the ending time is not known a-priori, requiring the capability to stop when some ending condition is met.

The simulation end can be achieved either by specifying a maximum time for the simulation (either in terms Virtual Time, or Wall Clock Time, depending on the simulation needs), or by checking if a particular condition in the simulation state is met. Time based ending is entirely managed kernel-side, while the second approach requires cooperation by the model, which can either be “asked” by the kernel to check whether the ending condition is met, or can actively communicate that its ending condition has been reached after handling an event.

Simulation Object. The concept of simulation object can provide interesting advantages to the modeller. A simulation object describes a portion of the whole model, whether it be a spatial portion, or an agent. Simulation Object is actually the name used to indicate Logical Processes. Thus, every Simulation Object is an LP, with its own state and events directed to it, and with the ability to raise events directed towards other LPs. The concept of simulation object is vital when moving from Discrete Event Simulation to Parallel DES.

3.2.3 DES Kernel Logic

Having introduced DES and its core components, and then having discussed the role of the kernel, this section delves into the fundamental logic behind a DES kernel.

As explained in Section 3.2.2, it is the kernel’s job—among others—to manage events and deliver them at the correct time instant. This process is kick-started by the scheduling of the *INIT* event after the kernel initialisation. A high-level flow of the kernel’s initialisation and event processing loop is shown in Algorithm 2.

Algorithm 2: Skeleton of DES execution. Source: [50]

```
procedure: INIT
    End $\leftarrow$ false
    initialize State, Clock
    schedule INIT
end procedure

procedure: SIMULATION-LOOP
    while End == false do:
        Clock $\leftarrow$ next event’s time
        process next event
        Update statistics
    end while
end procedure
```

3.3 Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) [19] refers to the execution of a single Discrete Event Simulation on a parallel/distributed system. This requires the DES kernel to be "augmented" into a PDES kernel implementation. As mentioned in Section 3.2.2, DES can be seen as two separate interacting components: the simulation kernel and the model. With this separation in mind, work can be conducted independently on the components to enact this transformation. Models can easily be ported to the parallel paradigm provided they respect a set of restrictions, the simulation kernel instead, has to undergo a series of important modifications. The advantage is that once the work on the simulation kernel is done, DES models can be adapted with minimal effort.

While in DES the concept of Logical Process described in the previous sections is optional, in PDES it becomes crucial. The name "Logical Process" reflects the parallel between LPs in simulations and processes in operating systems. Indeed, LPs operate independently from one another, and without sharing any memory. They thus have to exclusively rely on messages (i.e. events) Ffor inter-LP communication. A PDES simulation is then composed by N LPs, each of which has its own unique identifier number assigned in $[0, N-1]$. We denote the LPs as $LP_0, LP_1, ..., LP_{N-1}$. While in DES the concept of a single Global Virtual Time sufficed, in PDES each LP has its own private clock, referred to as Local Virtual Time (LVT). That of GVT remains a crucial concept in PDES, as it represents the time horizon for simulation commitment. The adoption of LVT means that two different LPs (e.g. LP_i and LP_j) can have different LVT values, $LVT_i \neq LVT_j$.

With the decoupling of LPs comes the first and most important model restriction: no shared variables may be used. This is because if two LPs, say LP_i and LP_j with different LVTs (suppose, without loss of generality, $LVT_i > LVT_j$) were to share a variable v, and LP_i were to write to v at LVT_i and Wall Clock Time t_i, and LP_j were to read v at LVT_j and WCT t_j, with $t_i < t_j$, there would be a causality violation, as the future affected the past, which violates the assumptions.

Thus, the modeller is required to split the simulation state S into per-LP sub-portions S_i such that:

$$S = \bigcup_{i=0}^{N-1} S_i \quad \wedge \quad S_i \cap S_j = \emptyset, \forall i \neq j \tag{3.1}$$

The restriction on global variables can pose no problem for some applications, but can also be troubling for others, forcing data organisation with sub-optimal performance, or requiring a great amount of messages to be exchanged.

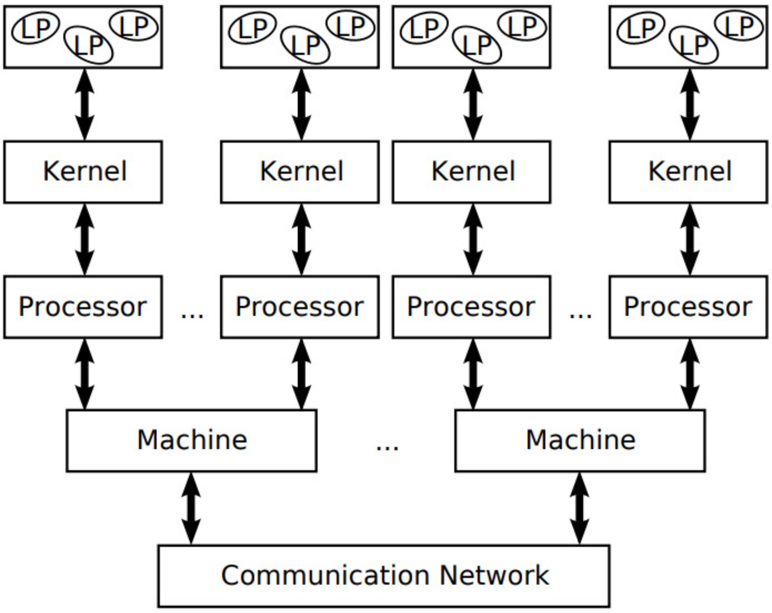


Figure 3.1. Classical architecture of a Parallel Discrete Event Simulator.
Source: [50]

In Figure 3.1 a classical distributed architecture of a PDES simulator is shown. Each LP is mapped to a simulation kernel instance, each of which can be seen as a user-space process, running on a processor. Different instances located on different machines are connected via a communication network. However, while Figure 3.1 can be a satisfactory abstraction, in reality the locality should be (and is) exploited by LPs hosted on the same machine, which can communicate through local inter-process communication facilities, lightening the burden put on the *distributed memory* that is in place to communicate with instances that are actually remote. In ROOT-Sim, such memory is implemented on top of message passing primitives. A message carries one event, as such there is a correspondence between message and event exchange.

In the last decade, the physical locality has been extended to kernel instances: while in the past each kernel instance would be a process run on a single-core CPU, nowadays the more lightweight concept of worker thread is used. The simulation kernel is still a process running on a CPU, but manages multiple processing units, and each worker thread runs the main simulation loop and is in charge of a number of LPs. This removes the inter-process memory separation, speeding message passing up between LPs sharing the same machine.

3.3.1 The Synchronisation Problem

In DES the execution was serial, posing no particular problems related to causality, only requiring a correct algorithm to pick the next event to be processed. In PDES, due to the parallel nature of the approach, extra steps are required in order for the absence of causality violations to be ensured. Let us take an execution (depicted in Figure 3.2) in which at a given WCT, LP_i (bound to thread k_0) has reached $LVT_i = 5$, and LP_j (bound to another thread, e.g. k_1) has reached $LVT_j = 16$. Since the two threads have extract events from different event queues, each of them will extract the appropriate next event according to Algorithm 2, obtaining events e_n^0 and e_n^1 as next events for LP_i and LP_j respectively. Suppose that $T_{e_n^0} = 10$ and $T_{e_n^1} = 20$. If during the execution of e_n^0 a new event e_{new}^0 with timestamp $T_{e_{new}^0} = 12$ and target LP_j were to be generated, then the situation would arise in which, while executing correctly, a LP_j would find itself at $LVT_j = 20$, having executed $T_{e_n^1}$, but with e_{new}^0—an event with a timestamp belonging in the virtual-time-past—in its event queue, creating a causality violation. The event that is "late" is called *straggler message*, consequence of nothing but the parallel nature of PDES. The fact that causality errors may arise because of the asynchronous nature of the execution, is the so-called *synchronisation problem.*

As it is inescapable, the synchronisation problem has to be overcome in order to obtain a simulation run that is perfect independently of the asynchronous nature of message exchange. Two main categories of approaches have been proposed: *conservative*, and *optimistic.* The conservative category avoids the occurrence of causality

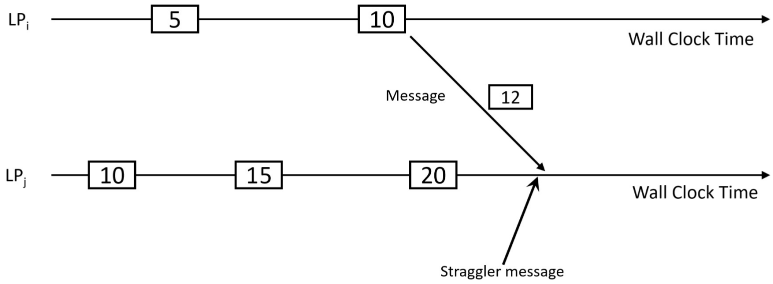


Figure 3.2. PDES execution with a straggler message.
Source: [50]

errors altogether by executing an event only once it is considered to be safe. Algorithms in the optimistic category (such as Time Warp [33], see Section 3.3.2) execute the next-to-be-processed event without any safety verification of causal consistency. Hence, timestamp-order violations might arise. If any such violation is detected, then correcting actions are taken. The two approaches are at the extreme and can be sub-optimal for a given workload, which gives rise to a third category: the *hybrid* approach, which tries to alternate between the two approaches to obtain the best performance. In this work, the focus will be directed towards optimistic synchronisation using the Time Warp algorithm.

3.3.2 Optimistic Synchronisation with Time Warp

In optimistic synchronisation, *speculative processing* is used: when it is unknown whether an event is safe to execute, it is executed in speculative fashion, the idea being that if no causality violation is detected, then the work done is committed and no computational resource was wasted waiting for the assurance of safety, while if a violation occurs, the additional work is discarded and correctness is preserved. The additional cost associated to correct guesses is none, while the cost associated to wrong guesses is equal to what a non-speculative system would have paid, plus the time needed to discard the work. This only takes into account time costs, while energy costs associated with the wasted computation are being ignored. This approach to tackling the synchronisation problem was first presented in [33], where *Time Warp* was introduced.

When a worker thread finishes executing an event and still has events in its queue, the next event to be executed is extracted and executed independently of its safety which, as we have seen when introducing the synchronisation problem, might send the execution into an inconsistent state, where during the execution a causal violation has taken place through a straggler message. If this happens, a rollback has to be executed.

Rollbacks. When a straggler message with timestamp $T_{straggler}$ arrives, a causality violation is detected. When this happens, the simulator needs to halt the execution, restore a consistent state whose VT precedes $T_{straggler}$, and restart the execution of the model. While this happens, any action taken when performing the computation that was discarded has to be reverted, including the sending of messages. This operation is known as *rollback* and was first introduced and popularised in database management systems to perform transactions in parallel.

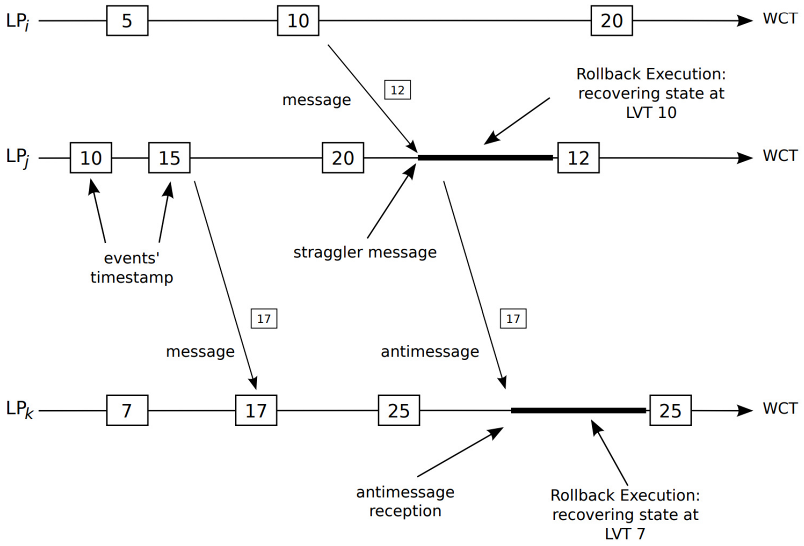


Figure 3.3. An execution in which a rollback takes place.
Source: [50]

When a rollback is performed, an additional problem may rise: during speculative execution, messages might have been sent. If any of these messages was created while executing an event that is getting rolled back, the message has to be reverted too. Let us see an example execution in which this could happen, in Figure 3.3. LP_j is at $LVT_j = 20$ when it receives the straggler message $e_{straggler}$ with timestamp $T_{straggler} = 12$. The execution is therefore rolled back to the last valid state: the one at $LVT_j = 10$. LP_j's state is now consistent again. However, during the execution of the event with timestamp $T = 15$, LP_j scheduled a message with timestamp $T = 17$ to LP_K. Since this message is the result of a processing that is being reverted, it has to be reverted, too. It is indeed possible that by executing $e_{straggler}$ the execution would follow an entirely different trajectory that never gives rise to the event associated with timestamp $T = 17$, or at least it does so with a different content. What happened is:

1. LP_i sends a straggler message $e_{straggler}$ to LP_j
2. LP_j detects the causality violation, rolls back the most recent consistent state preceding $T_{straggler}$
3. LP_j has to undo the sending of message with timestamp $T = 17$ to LP_k

The cancellation of messages is achieved through antimessages. An antimessage is a "negative copy" of a given message, signalling it has been undone. When rolling back, the LP checks if any message was sent during speculative forward execution, if so, the corresponding antimessage is sent to the same target. Upon receiving an antimessage, one of two things happen at the receiving LP LP_k:

- The antimessage has a timestamp $T_a > LVT_k$, which means that the message that the antimessage wants to annihilate is still in the Event Queue. This is the best-case scenario, as in this case the effect of the antimessage is limited the removal of the message from the queue, and thus low-impact.
- The antimessage has a timestamp $T_a < LVT_k$, which means that the positive message has already been processed. This situation is analogous to receiving a straggler message: LP_k needs to perform a rollback to the last valid state with $LVT_k < T_a$. This rollback could itself cause the sending of more antimessages, creating a chain of rollbacks.

We see how a causality violation can cause a series of rollbacks. This is referred to as *cascading rollback*.

The rollback operation can be executed in one of two ways: by checkpointing memory with *state save & restore*, or by *reverse computation.* When reverse computation is chosen, the model code needs to be able to reverse the effects of event handling to make the system travel back in time, adding computational overhead and hindering transparency. Furthermore computation might not be easily reversible. With checkpointing, the rollback can be executed in a completely transparent manner with respect to the model's point of view: the only prerequisite is for the kernel to know what are the memory areas that contain the LP's state. The kernel uses a series of data structures to create *snapshots* of the simulation state to transparently and safely revert it to a coherent one in case a causality violation were to present itself. Before delving deeper into the inner workings of *state save & restore*, another fundamental concept has to be introduced: Global Virtual Time.

Global Virtual Time (GVT). Introduced in [33], Global Virtual Time (GVT) serves multiple purposes when performing optimistic simulation. The GVT is calculated on a global snapshot of the system at WCT t. GVT(t) is defined as the minimum timestamp of any unprocessed message or anti-message flowing in the system at Wall Clock Time t.

The computation of GVT(t), is carried out by inspecting the timestamps of all the unprocessed messages currently in the system. Due to the distributed nature of PDES, this also includes the timestamps of messages that are currently being delivered by the messaging subsystem.

Given its definition, GVT lets us keep track of the *commitment horizon*: since in (P)DES, for the very concept of causality, no event e with timestamp T_e may schedule an event e' with timestamp $T_{e'} < T_e$, this means that, because GVT(t) is the minimum timestamp across all unprocessed events in the system $GVT(t) = T_{min}$, no event $\hat{e}$ such that $T_{\hat{e}} < GVT(t)$ will ever be scheduled. As such, at WCT t no straggler message could possibly exist that would cause any LP to rollback to $VT < GVT(t)$. We can thus say that all the events that have a timestamp $T < GVT(t)$ are *committed*, meaning that they will never be rolled back and can be used to perform I/O operations safely or verify the ending condition.

State Save & Restore. To support speculative execution, a checkpointing subsystem is essential. This subsystem periodically takes snapshots of each LP's simulation state. Should a rollback be triggered, the state of an LP can be restored to the a valid one using the most recent valid snapshot.

Supposing that a snapshot is taken after processing each event, when a causality violation is detected at time $T_{violation}$ and a rollback operation has to be carried out, it is sufficient to restore the state snapshot with the highest timestamp among the snapshots with $T_{snapshot} < T_{violation}$.

Taking snapshots is however a costly operation in terms of time and memory. As such, increasing the time that elapses between consecutive snapshots can save resources. This however has its own drawbacks, namely when a causality violation is detected with timestamp $T_{violation}$ and a snapshot with $T_s < T_{violation}$ is restored following the rollback, any event e whose timestamp T_e happens to be $T_s < T_e < T_{violation}$, its execution will have to be carried out again before continuing. As such, the snapshotting interval has to be chosen (either statically or dynamically) to be one that minimises the amount of time spent creating snapshots and re-executing correct events that were rolled back because of snapshot sparsity.

For memory usage, using GVT offers a solution: if a state has been committed it is guaranteed to be correct and will never be invalidated (i.e. rolled back). All the snapshots with timestamp prior to to the GVT ($T < GVT(t)$) can be safely discarded, as they will never be used to restore a valid state. Only the most recent committed snapshot always needs to be retained, to ensure a correct simulation state can always be restored. The process of discarding old snapshots and committed messages is called *fossil collection*.

3.3.3 Additional Supports for Simulation

PDES simulation kernels rely on a variety of elements to carry out their job, especially in an optimistic simulation context, where events must be stored (at least, those with a timestamp $T_e > GVT$) and snapshots taken. In Figure 3.4 a schema of a reference implementation including the various systems and data structures is provided.

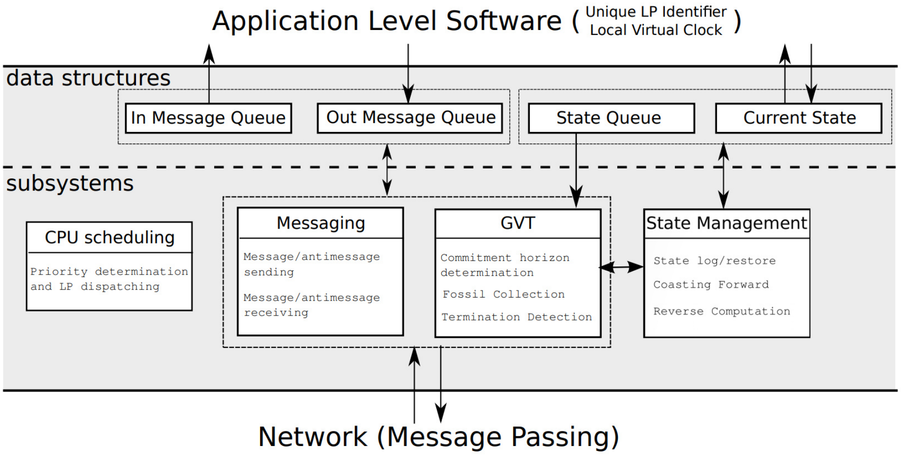


Figure 3.4. A Reference Architecture for Optimistic Simulation Systems.
Source: [50]

Input and Output Message Queues. While we have mentioned the (input) EQ earlier, more queues are needed to properly support optimistic simulation: a per-LP output queue needs to be maintained in order to generate antimessages in case of a rollback.

Messaging Subsystem. The messaging subsystem is vital as it lets the model be decoupled from the fact that the application is distributed. Indeed it takes care of message passing and the model relies on its APIs to perform the scheduling. It is then the kernel that takes care of where (and how, depending on whether the target LP is local or remote) a message must be delivered. Furthermore it can handle the output queue internally, which allows to decouple rollbacks and antimessage-sending.

State Queue and State Management Subsystem. The state queue is used to store the various snapshots that are kept by the system to be restored in case of a rollback. This subsystem takes care of:

- Maintaining a list of snapshots ordered by timestamp. When a new snapshot is taken, it is inserted in the list.

- Performing rollbacks by determining the correct state to be restored from the State Queue.

- Performing *coasting forward* (a.k.a. silent execution), that is, reprocessing the intermediate events between the restored snapshot and the timestamp of the straggler message, but without sending any messages (as no antimessages were generated for events causally dependent from such events, as they were correct, just not included in the snapshot).

- Performing fossil-collection, i.e. discarding messages and states with timestamps older than current GVT (except from the most recent snapshot as explained above).

GVT Subsystem. The GVT subsystem is in charge of computing the GVT at the scheduled time intervals, by accessing the message queues and the message subsystem. Furthermore it is tasked with termination detection, meaning that it either evaluates if GVT is greater than a given value, or by checking if a termination condition is met by evaluating some predicate. Lastly, fossil collection is also one of the subsystem's responsibility, by freeing old messages and logs that are unneeded because the related part of simulation has been committed.

Event Scheduler. Is tasked with deciding which is the LP (among those owned by a simulation kernel instance) that needs to be scheduled next. As mentioned, the typical approach is that of Lowest-Timestamp-First. This avoids the generation of causality violations among the LPs hosted by the same kernel.

Random Number Generators (RNG). In simulations relying on RNG, it is required that the generation be carried piece-wise deterministically. As such, when a rollback happens, the state of the RNG has to be rolled back as well. This can typically be done by storing the RNG status inside the snapshots.

3.4 Modelling Neurons for Parallel Discrete Event Simulation

Neuron models are commonly expressed as a set of differential equations that describe the behaviour of the membrane potential and synaptic currents. Given a

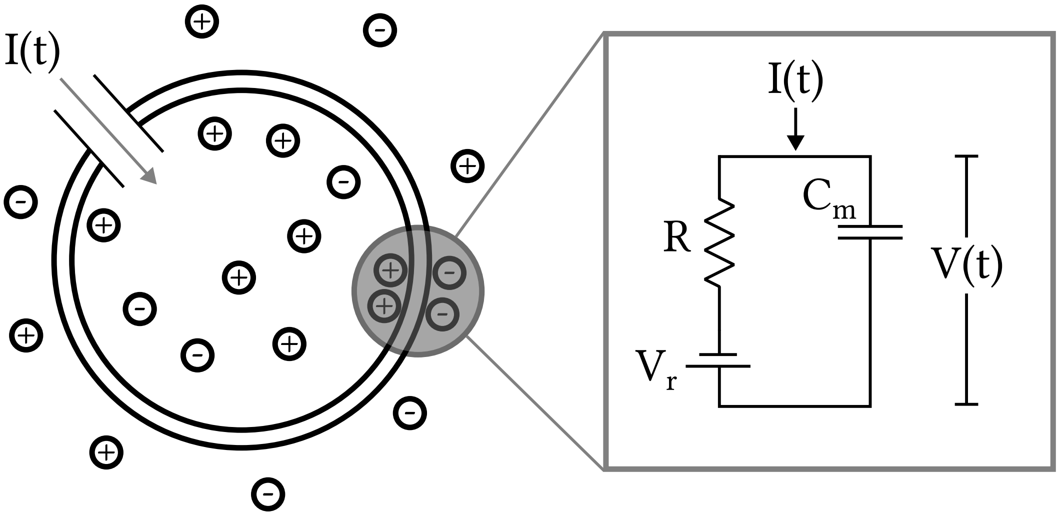


Figure 3.5. A neuron (on the left) is enclosed by the cell membrane (the circle). When it receives a positive input current $I(t)$, it increases the electrical charge inside the cell. The cell membrane is modelled as a capacitor in parallel with a resistor, which is in line with a battery of potential V_r.

known neuron state, generating discrete event spikes require computing the next spike timing. In the case of a Leaky Integrate and Fire (LIF) neuron (introduced in Section 2.2), the differential equations have an analytical solution, which we will hereby solve for a matter of precision and convenience. Other neuron models showing dynamics that cannot be expressed in analytical form, might still benefit from being simulated using PDES. For more on the topic, see Chapter 6.

While the modelling approach that we present is general, we exemplify it by showing how we can manipulate a neuron model to be simulated on top of a speculative PDES runtime environment adhering to the Time Warp synchronisation protocol.

We focus here on the LIF neuron model, which conveniently is also one of the most commonly used in large SNN simulations, with exponential current based synapses. For reader convenience, Figure 3.5 reports a scheme of the neuron and the circuit used to model its behaviour when using the LIF model. Equation (3.2) describes the sub-threshold dynamics of the neuron, that is, the evolution of its state in the absence of emitted spikes, where $V(t)$ is the membrane potential, and $I(t)$ is the incoming synaptic current flowing into the neuron. Equation (3.3) models the behaviour of an input synaptic current using an exponential synapse model (see Section 2.3.2). The two equations form a system modelling a LIF neuron with exponential synapses, and will be manipulated together to come to an analytical solution.

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = \frac{-V(t) + V_r}{\tau_m} + \frac{I(t) + I_{ext}}{C_m} \tag{3.2}$$

$$\frac{\mathrm{d}I(t)}{\mathrm{d}t} = -\frac{I(t)}{\tau_{syn}} \tag{3.3}$$

The positive quantities τ_m, τ_{syn}, C_m represent the membrane time constant, the synaptic time constant and the membrane capacitance, respectively. The quantity I_{ext} is a constant external current stimulus that is either fed to the neuron, or models the inputs coming from sources external to the simulation, while V_r is the reset potential. For a more thorough discussion on the meaning of these parameters, the reader can refer to Section 2.2 and to [9].

The non-linear spike behaviour works as follows: if, at any time t, $V(t)$ overcomes a voltage threshold V_{th}, the neuron emits a spike, then $V(t)$ is forcefully reset at voltage V_r for a period of time τ_r, the so-called refractory period. Spikes are delivered to post-synaptic neurons with a delay in virtual time and an effect established by the synapse model. Many large simulations employ static synapses, characterised by a fixed transmission delay t_{trans} and weight w. When using exponential synapses, a spike causes the post-synaptic neuron to instantaneously increase its $I(t)$ by w. Overlapping the effects of different synapses into a single incoming current value $I(t)$ is possible because the model for exponential synapses is Linear Time-Invariant (LTI).

Solving Equations (3.2) and (3.3) in $V(t)$ and $I(t)$ yields Equations (3.4).

$$\begin{gathered} V(t) = I_0 A_1 e^{-\frac{\Delta t}{\tau_{syn}}} + (V_0 - A_2 - I_0 A_1) e^{-\frac{\Delta t}{\tau_m}} + A_2 \\ I(t) = e^{-\frac{\Delta t}{\tau_{syn}}} I_0 \end{gathered} \tag{3.4}$$

In these equations, V_0 and I_0 represent the state of the neuron at time t_0, while the two constants A_1 and A_2 have been introduced for the sole purpose of readability: their expansions are found in Equations (3.5). Constants A_0 and A_2

can be computed once at simulation startup.

$$\begin{aligned} A_1 &= \frac{1}{\left(\frac{1}{\tau_m} - \frac{1}{\tau_{syn}}\right) C_m} \\ A_2 &= V_r + \tau_m \frac{I_{ext}}{C_m} \end{aligned} \tag{3.5}$$

To compute the next spike timing, it would be sufficient to solve Equations (3.4) in t with $V(t) = V_{th}$. Unfortunately, the analytical general case solution in t is yet to be found, as it is a transcendental equation with no direct method to solve it.

We must therefore resort to relatively expensive numerical methods. To minimise reliance on these methods, we now carry out further analysis that, while applied to this specific neuron model, has the potential to be applied to any model with analytical solution, by following the same method. We emphasise that the method is likely only applicable to models with analytical solution. Regardless, models without an analytical solution would not benefit from this reasoning, as their evolution still needs to be approximated and stepped through using numerical methods. For how using PDES could still benefit such models with respect to time-stepped simulation, despite the fact that they are stepped through, again we refer the reader to Chapter 6.

3.4.1 Predicting Spike Times in Analytical Models

It is possible to distinguish between self-spiking and non-self-spiking neurons. If, for a given neuron, its set of parameters is such that the condition in Equation (3.6) holds, the neuron is self-spiking. In other words, it spikes periodically on its own, without needing to receive any incoming spikes.

$$\lim_{t \to \infty} V(t) = A_2 > V_{th} \tag{3.6}$$

In this case, using the bisection method, we can compute once at startup the constant τ_{self}, the self spike timing in absence of synaptic inputs, i.e. $I(t_0) = 0, V(t_0) = V_r$. With t_0 indicating the current time in the simulation. To find a suitable window for the bisection method, we can simply consider the interval $[0, t_{large}]$ into the future, with t_{large} chosen larger than the time it takes for the neuron's membrane potential to reach V_{th} when $V_{t_0} = 0$ and no incoming spikes are registered. In other words, t_{large} is the time necessary for the neuron to self spike, starting from the resting state. Since Equation (3.6) holds by assumption, t_{large} must exist. On a practical level, it should be noted that in the case of inhibitory (negative) synaptic input, the neuron might take longer than t_{large} to spike, as waiting for the inhibitory inputs to wear off could be necessary. In this case, t_{large} still provides a reasonable starting point for a search window. Should $V(t_{large}) < V_{th}$, we can simply move the bisection window forward, from $[t_0, t_{large}]$, to $[t_{large}, 2t_{large}]$, to $[2t_{large}, 4t_{large}]$, and so on. Again, since Equation (3.6) holds and synaptic inputs always decay, a suitable upper bound t_{ub} in which $V(t_{ub}) > V_{th}$ will eventually be found.

For a non-self-spiking neuron, assuming the absence of incoming spikes, we derive a procedure to establish whether it will emit a spike in the future. In Equations (3.7) we provide the definition and the explicit value of the constant I_{th}. In a nutshell, I_{th} is the minimum synaptic input required for the potential to reach the firing threshold. Being $I(t)$ monotonic decreasing, $I_0 > I_{th}$ is a necessary (but not sufficient) condition for a non-self-spiking neuron to spike in the future.

$$\begin{aligned} I_{th} &:= I(t) \mid V(t) = V_{th} \wedge \frac{\mathrm{d}V(t)}{\mathrm{d}t} = 0 \\ I_{th} &= C_m \frac{V_{th} - V_r}{\tau_m} - I_{ext} \end{aligned} \tag{3.7}$$

If a non-self-spiking neuron satisfies the necessary spike condition, we can compute t_{th}, the time needed to decay from I_0 to I_{th}, using Equation (3.8). If $V(t_{th}) < V_{th}$ then we can conclude that the neuron does not spike, otherwise it will spike at a time $t_{spike} \in [t_0, t_{th}]$.

$$t_{th} = -\ln\left(\frac{I_{th}}{I_0}\right) \tau_{syn} \tag{3.8}$$

This result holds because we have $V(t_0) < V_{th}$ and $V(t_{th}) \geq V_{th}$ therefore, given the continuity of the $V(t)$, at least one $t_{spike} \in [t_0, t_{th}]$ must exist. Otherwise, if $V(t_{th}) < V_{th}$, then $V(t) < V_{th}$ for all $t \in [t_0, t_{th}]$: the neuron did not and will not spike in absence of further stimuli since $I(t) < I_{th}$ for all $t > t_{th}$. In the case in which we know the neuron will spike, we use the bisection method to compute t_{spike}. Bisection uses a starting search interval $[t_{min}, t_{max}]$—in this case $[t_0, t_{th}]$—and narrows it down by iteratively splitting it in half and picking one of the two halves—the one holding t_{spike}—, and discarding the other. The algorithm stops when the diameter of the search interval is less than or equal to an arbitrary value, which we refer to as ROOT-Sim 's "tolerance", "precision", or "resolution": $t_{max} - t_{min} \leq \textit{tolerance}$. Higher order numerical methods were briefly explored with an implementation of Newton's method. However, after observing numerical stability issues, it was decided to focus on the development of runtime supports, and the usage of more complex numerical methods postponed to future work.

The proof that t_{spike} is unique in the $[t_0, t_{th}]$ interval is more involved, so we will only sketch the main idea. Using the definition of I_{th} it is possible to show that $\frac{\mathrm{d}V(t)}{\mathrm{d}t} = 0$ for exactly one $t_d \in [t_0, t_{th}]$, where $V(t)$ reaches its maximum. We have that $\frac{\mathrm{d}V(t_0)}{\mathrm{d}t} > 0$ and $\frac{\mathrm{d}V(t_{th})}{\mathrm{d}t} \leq 0$. Also, since $\frac{\mathrm{d}V(t)}{\mathrm{d}t}$ is continuous, we can conclude that $V(t)$ is monotonous increasing in $[t_0, t_d]$ and monotonous decreasing in $[t_d, t_{th}]$; t_{spike} is therefore unique and lies in the interval $[t_0, t_d]$.

Regardless of the neuron type, the future spike time will ultimately be found. The neuron then tells the simulation kernel it will spike at t_{spike}, if no other spikes are delivered to it. The facilities to support this capability are presented in the upcoming sections.

3.4.2 Managing Spike Events

Spikes are represented in the simulation kernel as messages, which are delivered to the destination LP at the time they need to be applied. Since a single neuron can be connected to a multitude of neurons, injecting one spike event for each destination LP every time a new possible spike time is calculated, will easily thrash the simulation due to the event management overhead.

A simple solution at the model level could be to inject in the system *tentative* spikes, i.e. events associated with some per-neuron epoch counter that ensures their validity. Every time the neuron state is updated due to the receipt of an incoming spike, the epoch counter would be increased, the new spiking time could be re-computed and a new spiking event (superseding the previous one) with the new

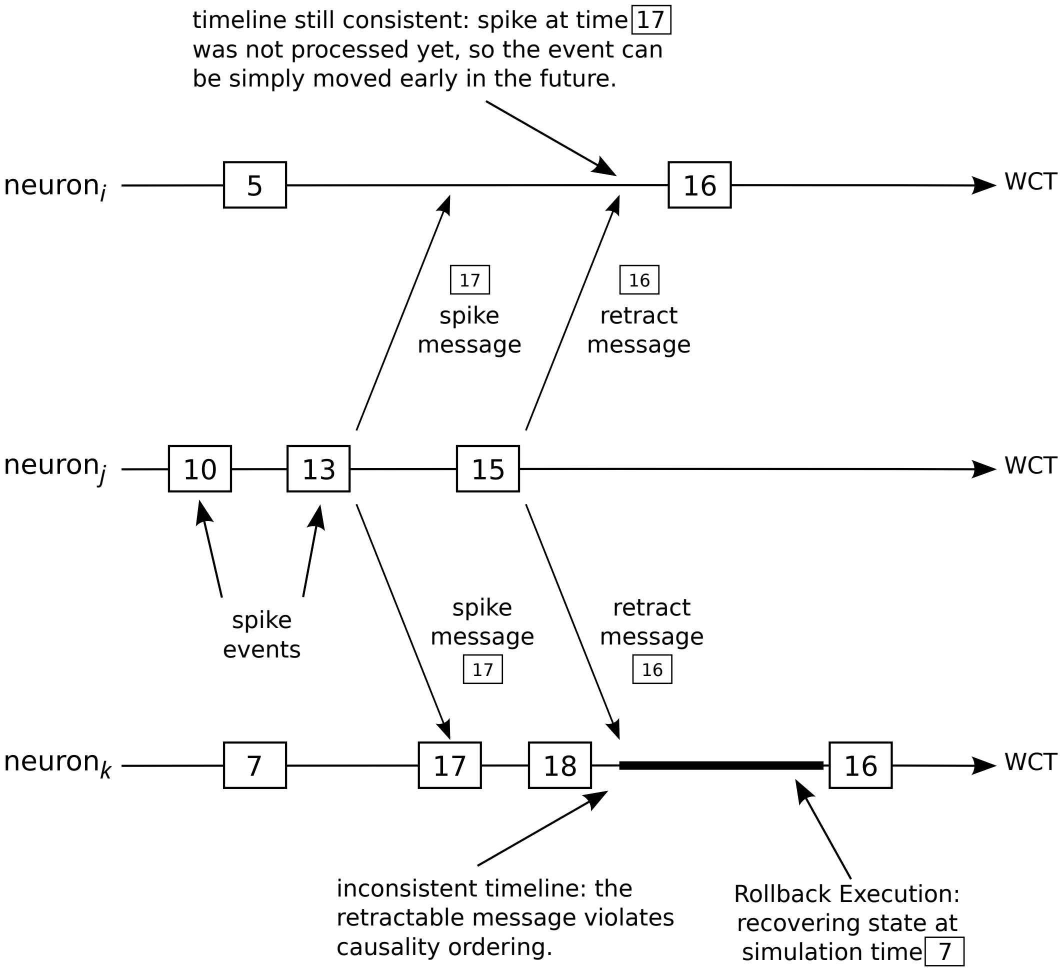


Figure 3.6. Retractable Spikes Naïve Scheme. A neuron can decide to change the time of a spike already injected in the system. Receiving neurons might have to rollback part of their execution.

epoch counter could be injected into the system. At the time of handling, a spiking event with a mismatching epoch counter would be considered stale. While this approach limits the generated events to one per received spike—and thus, updated future spike time—, it is unlikely to scale due to the large amount of extremely-fine grained simulation events produced, and that are typically the cause of poor performance in Time Warp simulations [20]. The performance degradation of this scheme stems from the strict decoupling between the model and the runtime environment in Time Warp simulations. In this scenario, the model cannot inform the runtime environment that a tentative spiking event should be removed from the system, and is thus forced to only logically discard it once it is delivered for execution.

For this reason, in [53] we introduced in ROOT-SIM the concept of *retractable events*, i.e. events that can be rescheduled and descheduled by the model. Logically speaking, this support allows implementing tentative spike events, according to the scheme depicted in Figure 3.6. After sending its tentative spikes to the receiving neurons as retractable messages, the “spiker” neuron n_s can reschedule the events via dedicated APIs, to effectively update the delivery time of said spikes. Take, e.g., the case of new excitatory spikes being delivered to n_s, which consequently charges faster, thus reaching the threshold V_{th} earlier. In this case, the neuron can inform the receiving neurons that the spike should be dealt with earlier. At the destination neurons, if the involved spike has not been processed yet, the event is simply moved earlier in the future. Conversely, if it has already been processed, a traditional rollback operation will restore the neuron state to a consistent timestamp, and the new spiking time will be considered in the simulation. This naïve approach still suffers from the fact that (1) a large number of control messages need to be scheduled (and then ignored) when implementing this approach and (2) the total number of rollbacks would likely remain high, crippling the simulation speed. An experimentation showing the cost of the approach can be found in Section 3.5.2.

A straightforward optimisation is to have every neuron deal locally with retractable events. Indeed, a neuron can determine its next firing time and schedule a *tentative spike-firing event* directed to itself. This tentative firing event is managed the same way as a regular firing event (i.e., the neuron sends the spikes to all destination neurons upon receiving it) if no change in the firing time occurs. Differently, if the neuron model determines a new timestamp for the firing event, the runtime environment will act accordingly on the message queue. In particular, the firing event will simply be moved to the newly calculated future firing time, if any, or descheduled. This way, the number of events injected into the system and the total number of rollbacks are significantly reduced, as the destination LPs will only receive a spike at the accurate firing time, after that the firing neuron has correctly received all pre-synaptic stimuli. All of this is achieved in a manner that is transparent for the model, as it is hidden behind a neural simulation interface (see Section 3.5.1). The implementation of dynamic spiking events as a runtime support and the benefits coming from it are dealt with in Section 3.5.2.

Having solved the issue of future spiking events thrashing the simulation, another important aspect remains to be explored: spike event delivery. In large simulations, a single neuron will be connected to a multitude of postsynaptic neurons: when gathering performance data, we ran networks where neurons had 6,000 connected postsynaptic neurons, on average. Since one event has to be generated per postsynaptic neuron (because of dedicated synaptic weights and transmission delays), the message passing subsystem can experience noticeable load every time a spike is propagated, or rolled back. This becomes more evident when running in a distributed environment, having to send thousands of events over the network, per spike.

To overcome this limitation, a lightweight implementation of Publish/Subscribe message dissemination algorithm was used, which extremely limited the amount of data transferred from worker thread to worker thread. More on the Publish/Subscribe message dissemination as a runtime support in Section 3.5.2. Again, this facility was used transparently for the model, thanks to the neural simulation interface.

3.5 Runtime Support

3.5.1 Neural Simulation Interface

To help modellers more easily develop Spiking Neural Network models using PDES, a **neural simulation interface** was developed, which takes care of interfacing with the simulation kernel, hiding its complexity from the modeller. The interface takes care of managing the memory holding the neuron state (which is controlled by the modeller) and the incoming synapses. Furthermore, it introduces a series of APIs that guide the model's execution flow. This is a lower level interface with respect to the one presented in Section 4, and allows new neuron and synapse models to be implemented.

While the technical details are lengthy and belong in a design document, a brief rundown of the functionalities is hereby presented. The interface exposes APIs for the model to:

1. **Spike** at a specific time, transparently delivering the spike to all postsynaptic neurons.
2. **Tentatively spike** at a specific time, only if no other events are received by the neuron in the meantime, and wake the neuron after spiking.
3. **Connect** two neurons via a synapse with specific weight and delay[1].

In turn, it expects the modeller to provide handlers for the following events:

1. Neuron initialisation - to initialise the neuron state and network topology.
2. Spike delivery - to allow the neuron to react to an incoming spike. The neuron is expected to communicate the updated spiking time every time this handler is invoked, or the spike is considered descheduled.
3. Neuron awoken after spiking - to allow neuron state to be updated, and any future spike to be scheduled.
4. Alignment event at simulation end - because there is no guarantee of LVT being aligned across different LPs (see Section 3.3), this gives the modeller

[1] Only static synapses are currently supported

the opportunity to align the states to the desired end time and even perform output[2].

The neural simulation interface uses the other runtime supports to ensure a faster execution. Retractable events are used for tentative spiking events: whenever a new tentative spike is scheduled, the interface actually reschedules the self-directed spike event to the newly selected time. If no tentative spike event is scheduled, any previously existing spike-firing event is descheduled. When a spike-firing event is extracted, the message is disseminated, and then the neuron woken up.

For message dissemination, upon creation of a synapse, the neural interface subscribes the postsynaptic neuron to the topic associated to the presynaptic neuron, ensuring delivery is handled by the simulation kernel. When a spike-firing event is extracted, it uses the Publish/Subscribe primitives to disseminate it to all the connected (i.e. subscribed) postsynaptic neurons.

3.5.2 Dynamic Spiking Events using Retractable Messages

We have shown earlier how to determine the next spike timing t_{spike} for a LIF neuron with the condition that, in the meanwhile, such neuron does not receive any spike. We also showed how it is possible to implement tentative spiking events without recurring to retractable events, by using an epoch counter, and a large amount of ignored messages.

In a discrete event simulation, we would schedule a new event e with time t_{spike} which represents the spike potentially emitted in the future. If the neuron receives a spike before t_{spike}, its previously computed spike timing would not be consistent anymore with the newly induced state change, and therefore we would need to somehow invalidate or retract the event e.

Our proposed solution consists in providing the model developer with *dynamic spiking events*, that is, with the possibility of scheduling *retractable events*: events that can be arbitrarily removed from the events queue or whose timestamp can be

[2] While no actual facility has been implemented yet to safely perform output, since even the alignment event is speculative, expedients are possible to correctly perform output. Implementing output in the interface is part of future work.

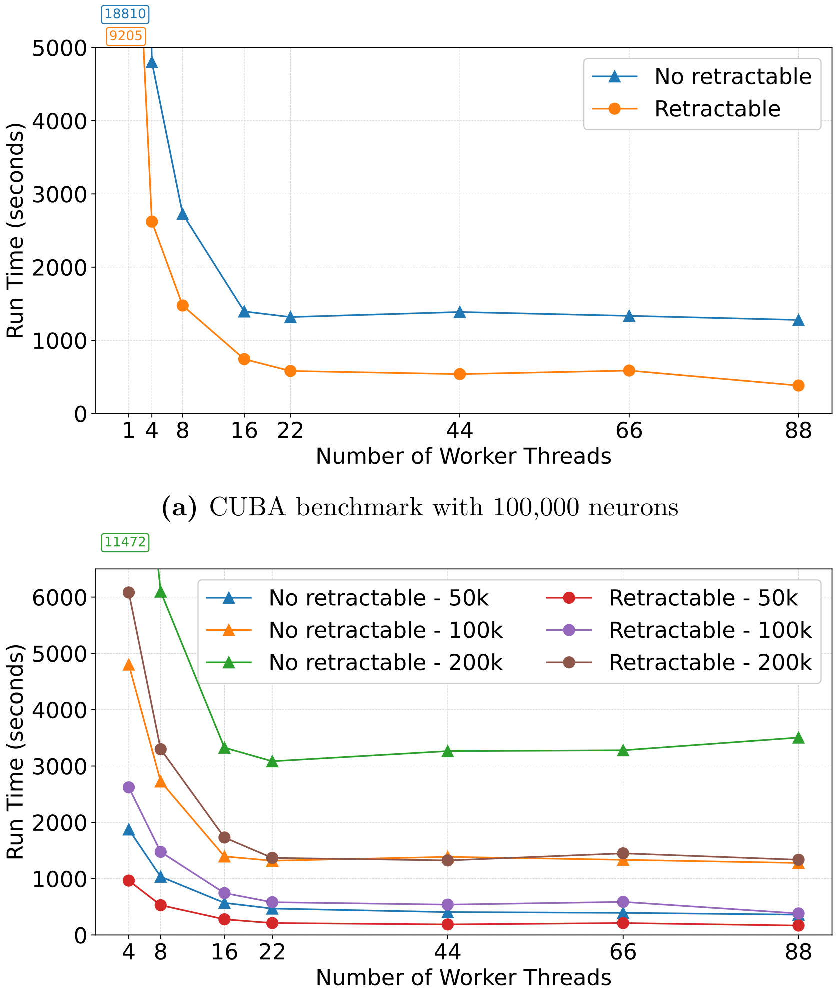
changed at will, with these operations supported at the runtime environment level. Future spike emission events can be effectively represented as retractable events.

In the context of SNN simulations, each neuron needs a maximum of one retractable event destined to itself. The spiking event has no payload: because of the way the neural interface is implemented, the fact that it got extracted for handling is enough guarantee that the scheduling neuron did not receive any spikes since the last time the spiking event was scheduled. These considerations significantly simplify the implementation of this new mechanism. Each thread simply has a private queue that handles the retractable events associated with the neurons bound to it. The private queue is implemented as a k-heap data structure which allows efficient priority changes and removal of events. In order to extract a new event, each thread chooses the lowest timestamp between the normal events queue and its private retractable events queue.

In order to correctly handle rollbacks, when computing the checkpoint of an LP, we must also include their currently active retractable event timestamp, and upon finishing the rollback, the event has to be rescheduled for the stored time. With this precaution, when an LP must roll back to a previous state, its retractable message is correctly restored, too.

In our experimentation, moving from utilising the naïve epoch counter approach to retractable events reduced running times by 50% in the worse cases, and by over 60% when using higher thread counts. Figure 3.7 reports the running times for the CUBA benchmark (see Section 1.2.2) when using retractable events, compared to not using them. The experiments were run on a machine with two Intel Xeon CPUs E5-2696 v4 @ 2.20GHz, with 22 physical cores and 44 virtual cores each, for a total of 44 physical and 88 virtual cores. The machine has 256GB RAM. Figure 3.7a reports times for a network of 100,000 neurons, using 1 to 88 worker threads, while Figure 3.7b shows different network sizes, using 4 to 88 threads: single worker thread runs were skipped for a matter of execution times. All data points are the average of 5 runs. For the sake of readability, data points that would have flattened the plots have been reported in numbers above the plot.

Retractable events are also helpful for other simulation models: for example,



(a) CUBA benchmark with 100,000 neurons

(b) CUBA benchmark varying network size

Figure 3.7. Run times when using the naïve spike scheduling method, compared to using retractable events. For the sake of readability, run times that would have flattened the plot, are indicated with a label outside the plot area.

they can be employed in Blockchain simulation to efficiently implement future block mining events that may be invalidated by receiving a new block or update to the blockchain.

3.5.3 Publish/Subscribe Event Dissemination

While the use of retractable messages effectively reduces the number of events dedicated to spike management, it has no impact on the number of events used to convey the actual spikes. Each time a neuron spikes, one event per connected postsynaptic neuron has to be generated and delivered. In larger models such as the CUBA model (see: Section 1.2.2), with 300,000 neurons and an average out-degree of 6,000 per neuron, this translates to a small "message-bomb" that can be cumbersome to disseminate across processing threads because of synchronisation, let alone over the network. An early experimental assessment showed that this was a significant performance bottleneck.

To address this, we extended the runtime environment with a feature that supports a light publish/subscribe event dissemination. To give an idea of the capabilities of the pub/sub runtime support using the terms of actual publish/subscribe messaging frameworks, every LP has its own topic, which is the only topic it can write to, and in which it is the only writer. Any LP can subscribe to any LP's topic, to receive messages published by the topic owner. For simplicity, given the absence of ambiguity, we say that *a is subscribed to b* to indicate that LP a is subscribed to LP b's topic.

The network topology is initialised during simulation initialisation by having LPs perform subscriptions to the other LPs. For SNNs, when neurons invoke the connection APIs to connect, e.g. n_s to n_d, the neural simulation interface actually subscribes n_d to n_s. This way, when n_s schedules a tentative spike event which is then extracted to be handled (i.e. the spike actually happens), the spike is emitted and propagated in the form of a published event. With this contract in place, the simulation kernel can significantly reduce the number of events actively transmitted.

To simplify the description of our implementation, we say that a processing thread is subscribed to an LP if any of the LPs bound to it are subscribed to

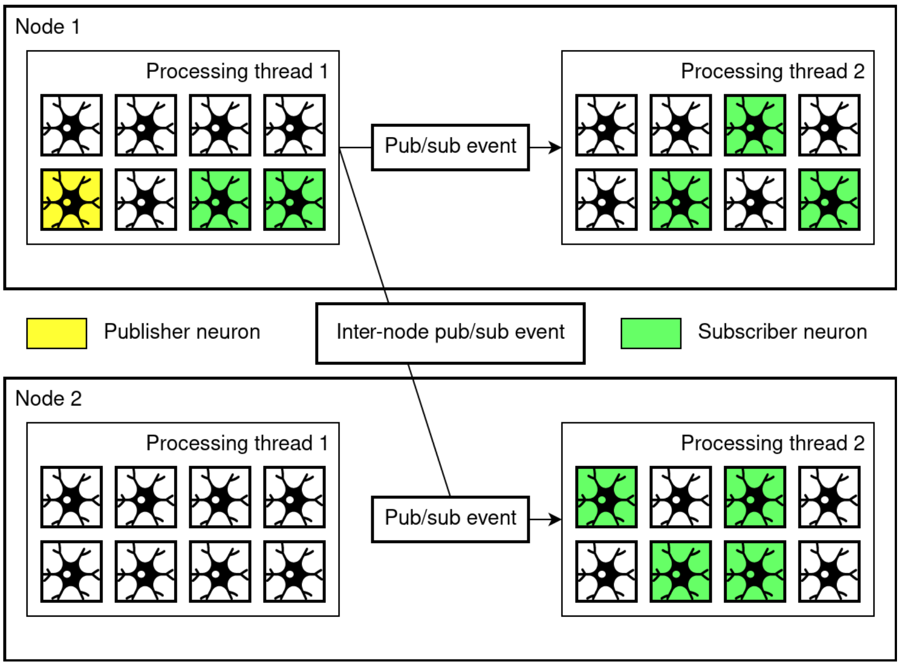


Figure 3.8. Publish/subscribe event dissemination in a distributed environment. In this example, the publisher LP generates only two events, instead of nine.

that LP. Similarly, a computing node is subscribed to an LP if any of the hosted processing threads is subscribed to that LP. Our implementation expects a single function that specifies the publish/subscribe graph. Each processing thread involved in the simulation initialises the internal data structures based on the relevant subset of information in the graph. In particular, each LP maintains a list of references to the local processing threads and remote nodes subscribed to it. Also, each node hosts a global table that maps identifiers of publisher LPs to a list of references to the local processing threads and related LPs subscribed to it.

As exemplified in Figure 3.8, publish/subscribe events can drastically reduce the number of events exchanged by processing units. When an LP publishes a new event, the simulation framework only generates and delivers the events as specified in its subscription list, sending one copy per local subscriber thread and one per remote subscribed node, minimising the number of messages transferred, both locally and over the network. The support implements a hierarchical structure for message delivery: inter-node messages, thread-level messages, and normal LP-level messages. Locally, a thread-level copy of the message is enqueued into each subscribed thread's incoming event queue. The same happens at other nodes, where the thread extracting the publish/subscribe event from the message-passing layer proceeds to look-up the threads subscribed to the sender LP from the global table, then creates and enqueues copies of the publish/subscribe event into their queues as a thread-level publish/subscribe event. When a processing thread extracts a thread-level publish/subscribe event from its queue, it simply needs to get the list of bound subscribed LPs from the global table, and forward them a LP-level copy of the event, by inserting the copy into the queue. The LP-level event is treated as a regular event for all matters. All the look-ups are $\mathcal{O}(1)$ thanks to the global map's structure, and the fact that all subscribed threads and LPs need to receive a copy in any case.

The management of the rollback operation for publish/subscribe events can be realised according to the traditional scheme supported by Time Warp synchronisation, i.e. by relying on anti-events. Indeed, after a single publish/subscribe event is materialised into multiple copies for each subscribed LP, publish/subscribe anti-events are treated no different from regular events.

To support anti-messaging, when an event is published, the message keeps references (pointers, actually) to all locally generated thread-level messages, so as to avoid having to look them up in data structures. The same happens for LP-level messages generated when handling a thread-level publish/subscribe message. During anti-messaging, the local message references are iterated and those not yet processed are annihilated, while the processed ones are re-queued as anti-messages in the respective threads. When a thread-level anti-message is extracted, the thread uses the list of pointers to generated messages to either annihilate them, or re-queue them as anti-messages.

To perform anti-messaging of remotely sent messages, negative copies are generated and sent. At the receiving node, when the worker extracting incoming messages from the message passing layer extracts a publish/subscribe anti-message, a thread-level copy of it is simply enqueued into relevant threads' queues, in the exact same fashion as positive messages. Once a thread extracts the thread-level anti-message of a remote-generated publish/subscribe event from its queue, it uses a thread-local hashmap to look-up the positive message (before processing a positive message generated from a remote node, a reference to it is added to the hashmap, and it is checked whether the negative version of the message was already received). If not found (i.e. the positive message was not processed yet), the anti-message is kept in the map to annihilate the positive message when it comes. If found, the anti-messaging is carried out the same way as above, using the list of pointers to generated messages to either annihilate or re-queue the messages as anti-messages.

All the used data structures are regularly fossil-collected with the rest of messaging subsystem to remove committed publish/subscribe events.

As far as performance is concerned, publish/subscribe event management provocated around 1% loss in performance in single node executions, due to the message unpacking overhead. However, the runtime support proved vital when executing multi-node simulations, drastically reducing the network traffic of the simulated models, reaching estimated reductions of up to 99.4%—meaning the execution used 0.6% of the bandwidth it would have used otherwise—on the CUBA model (see Section 1.2.2) using 300,000 neurons and 32 processing nodes. The bandwidth reduction is proportional to the number of connections between Logical Processes, and inversely proportional to the number of computational nodes.

Chapter 4

Programming Interfaces

As stated various times in this work, as a method, speculative PDES allows for efficiently executing simulations by partitioning the simulation space into smaller simulation objects, each managed by a separate processor or computing node. Despite being scalable while maintaining high accuracy, the expertise required to implement effective SNN models on top of speculative PDES can limit its adoption among researchers and practitioners.

State-of-the-art SNN simulators, such as NEST, NEURON, and Brian, provide both native Python interfaces and PyNN backends allowing easy model creation, execution and comparison.

Listing 1. A minimal example of SNN simulation using PyNN

```
sim, options = pyNN.utility.get_simulator()
sim.setup()
pop = sim.Population(50, sim.IF_curr_exp())
pop.initialize()
rng = sim.NativeRNG(seed=0)
conn = sim.FixedProbabilityConnector(0.5, rng=rng)
syn = sim.StaticSynapse(weight=0.02)
sim.Projection(pop, pop, conn, syn, receptor_type='excitatory')
sim.run(1000)
sim.end()
```

Following this approach, to overcome these challenges and make SNN simulation based on speculative PDES more accessible, we also proposed a novel, user-friendly approach based on PyNN [16], a widely-used neural network modelling library.

PyNN provides a high-level, unified, and standardized programming interface for defining neural network models and running simulations on any of the available simulator backends. Among the supported simulators we find NEST [21], NEURON [31], and Brian [64].

By leveraging PyNN, our approach aims to simplify the deployment of SNN simulations on top of speculative PDES runtime environments.

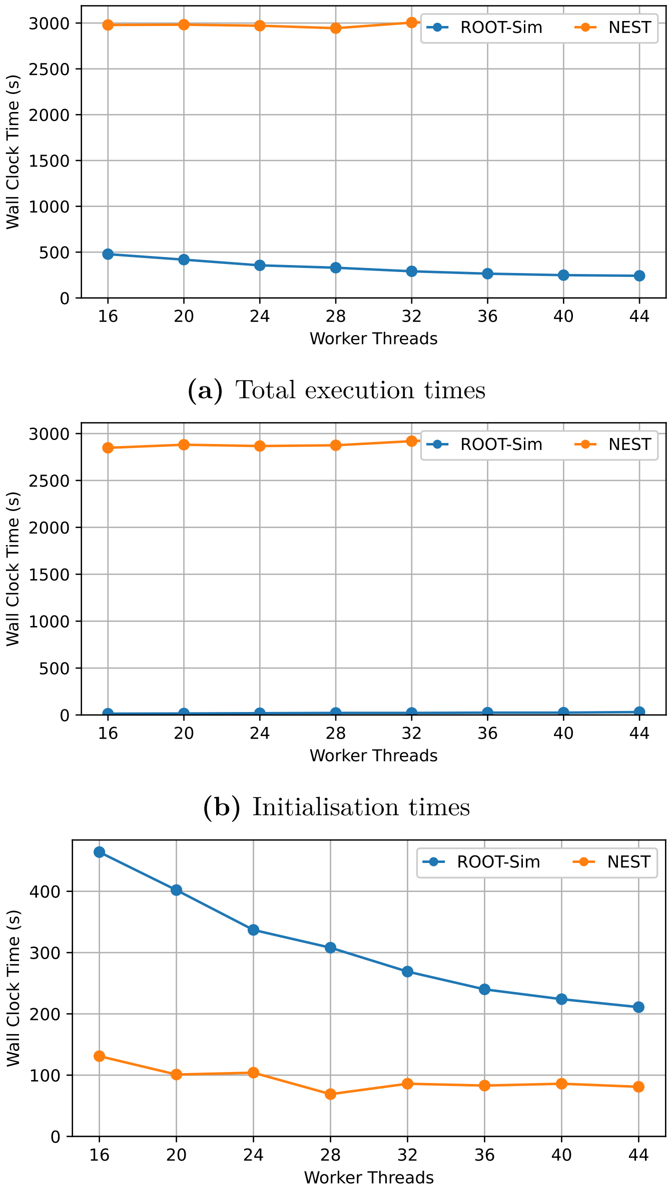
We hereby present a minimal implementation of a PyNN backend, which encapsulates the SNN simulation interface for ROOT-Sim, providing a user-friendly interface for configuring and deploying PDES-based SNN simulations. We also evaluate the overhead introduced by the additional layer through a benchmark simulation.

Listing 1 contains a Python snippet using PyNN. It gets the simulation backend and command line arguments, initialises the simulator, a neuron population of size 50, and the random number generator. Creates a random connector with a pairwise connection probability of 50%, and connects the neuron population to itself with static excitatory synapses. Then simulates for 1 second. While very essential, this code is a perfect testament to how straightforward model development becomes using these interfaces.

4.1 Implementation

Implementing a PyNN backend means implementing a Python class that exposes the PyNN APIs in compatibility with the simulation framework that will, in the end, run the simulation.

In our implementation, the Python interface does not interact directly with ROOT-Sim, but rather, it responds to the various API calls by writing to its internal state. Once the run command is given, only then does the interface start the interaction with the simulation backend. The interface generates a C source and header file couple with specific names, which are then compiled against an ad-hoc



(a) Total execution times

(b) Initialisation times

(c) Simulation times

Figure 4.1. Run times using PyNN interfaces

SNN model for ROOT-Sim. Once the compilation is completed, the Python class will launch the resulting binary, that runs the simulation.

The SNN model implements the main function, and upon launching takes care to initialise the topology as specified in the PyNN-generated files. The topology can be specified explicitly, or the fine details can be delegated to the simulator, which takes care of connecting the neuron populations according to the specifics provided.

If the requested connection pattern is probabilistic, the simulator can use its internal random number generation facilities to build the connections, drastically reducing the amount of computation carried out in the Python script, which speeds up the initialisation, especially in the case of large networks.

Before initialising the topology, the ad-hoc model instantiates the neurons and their initial state, either to a default state, or randomically, or by reading from the generated files. The topology is then materialised by connecting neurons to one another. Finally, the simulation is launched and executes until the specified end time is reached. The model can produce output to file, to track the generated spikes.3

4.2 Evaluation

The new PyNN backend communicates with ROOT-Sim via a generated configuration file from which the simulator reads topology and initialisation information. To evaluate the overhead introduced by this approach, we ran the CUBA benchmark (presented in Section 1.2.2) using the PyNN interfaces. This version of the model has a network of 100,000 neurons, 80,000 excitatory and 20,000 inhibitory. The connection probability of 2% leads total of ∼200,000,000 synapses.

The experiments were run on a machine with two Intel Xeon CPUs E5-2696 v4 @ 2.20GHz, with 22 physical Cores each, for a total of 44 physical Cores. The machine has 256GB RAM. Each data point represents the average of three runs.

The PyNN backends we compared are (1) the one which is currently being presented, encapsulating ROOT-Sim, and (2) the one for the NEST simulator, provided in PyNN.

The network topology is simple, thus easy to define using PyNN. When available, a backend-provided Pseudo-Random Number Generator (RNG) was used for neuron initialisation and topology definition to maintain the highest possible performance.

Because of the fast-paced development of SNN simulators, the native RNG implementations of both NEST and NEURON simulators were unusable, due to missing or outdated components, which would raise errors and result in the script exiting. This forced the usage of a NumPy-based RNG that PyNN provides, which, however, carries out the entirety of the pseudo-random number generation within Python, drastically increasing initialisation times (see Figure 4.1b). Initialisation times using NEST's built-in RNG with PyNN are thus unknown.

This obviously gave the ROOT-Sim implementation a noticeable edge in end-to-end execution times (Figure 4.1a). For this reason, the more insightful plots of *initialisation time* (Figure 4.1b) and *simulation time* (Figure 4.1c) are also provided.

It is interesting to notice how the overall performance of ROOT-Sim is essentially unaffected by the presence of the Python interface, thanks to the possibility of providing its internal RNG.

The simulation performance is in line with that observed in our other works (see, e.g., [54, 53]), and that will be presented in Chapter 5. When comparing the execution of the model by the two simulators, we see how ROOT-Sim starts slower but benefits more from the added worker threads, while NEST starts with faster times, but extracts less performance from additional computational power.

The presented data shows how exposing adequate interfaces renders the performance overhead negligible, while making the cost of adoption aligned with that of existing time-stepped simulation tools, for which various PyNN models are available.

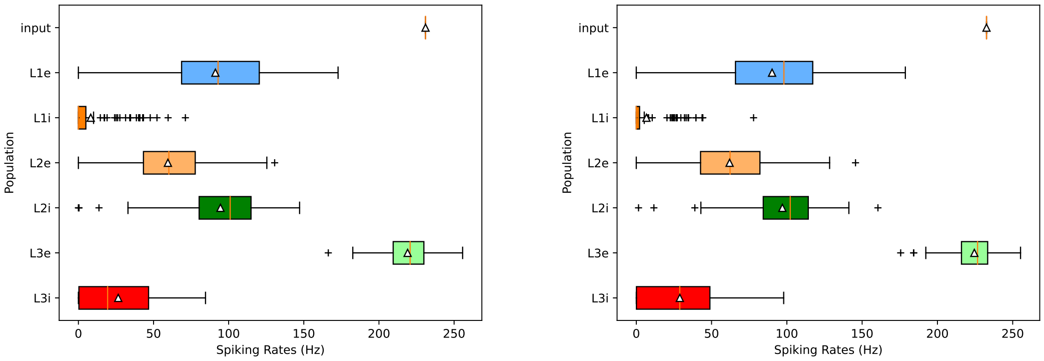
Chapter 5

Experimental Evaluation of Simulation Performance and Accuracy

In this chapter, various results of the conducted research are presented. First we conducted experiments to ensure the adherence of produced simulation results to expected behaviours, comparing with other simulators. Then a performance and scalability evaluation is presented. Thirdly, we present our findings regarding precise accuracy of SNN simulations using PDES and Time-Stepped approaches. Finally, we recreated a network topology used to perform classification tasks and evaluate the prediction accuracy, and present the findings.

5.1 Statistical Verification

Firstly, in our experimentation, we performed statistical verification of the implementation. This is done by picking a model, executing it, and then comparing the neuronal ensemble behaviour to the one reported by other simulators, using the distributions of spiking frequencies. Due to the different nature of implementations, this was intended as a rough correctness check of our implementation, before moving on to more detailed exploration as presented in Section 5.3, which will play an important role in the explanation of the data we are about to present.



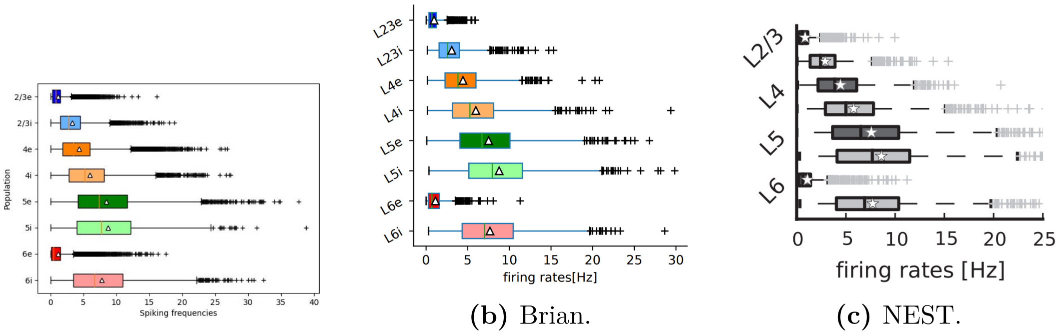
(a) ROOT-Sim.

(b) Brian.

Figure 5.1. Spiking rates (Hz) for 1000 neurons model.

1000 Neurons. First we start with the simple synthetic 1000 neurons benchmark, presented in Section 1.2.2. This small model was deemed ideal to verify the ensemble behaviour of the network in a contained way, while still in presence of various differently timed spikes. We implemented the same network on both our simulation framework, and on the Brian SNN simulator (see Section 1.2.1). We then ran the network and collected the spiking rates from each neuron. The box-plots produced from the executions are reported in Figure 5.1. It is clear from a quick look at the figure, that the box-plots of spiking frequencies closely resemble one another. The difference was attributed to the randomised initialisation of the network topology, with the small size of the network not being enough to amortise randomness, and some difference introduced by the limited precision Brian had due to time-stepping, as it did not use closed form equations for neural dynamics. This last observation would then be further corroborated by the additional knowledge regarding simulation accuracy that we gathered with successive study and experimentation, the results of which are presented in Section 5.3. Nonetheless, due to the contained nature of the observed discrepancies, this verification showed how the framework and LIF model implementation of the neurons are correct even when a modest number of neurons is simulated, and with neurons handling spikes coming in varied patterns from many other neurons.

Local Cortical Microcircuit Model. We have also considered the Local Cortical Microcircuit specification by Potjans and Diesmann (see Section 1.2.3) to confirm



(a) ROOT-Sim.

(b) Brian. Source: [62]

(c) NEST. Source: [58]

Figure 5.2. Spiking rates (Hz) for Local Cortical Microcircuit model.

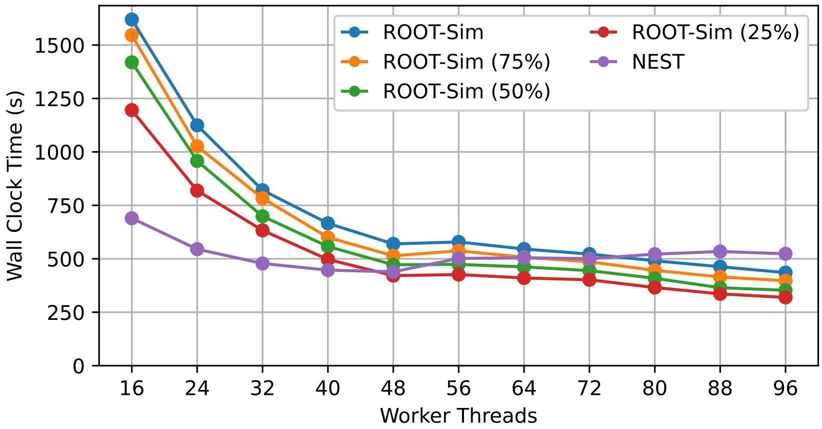
the results above. Again, this model is highly relevant, with its implementation being part of the examples in the NEST simulator (see Section 1.2.1) exhibiting similar behaviour to the one observed in-vivo. The model has also been replicated in 2018 in [62] with an implementation for the Brian simulator. We have compared our implementation of this model with the implementations in both NEST and Brian.

The simulation results are shown in Figure 5.2, again in the form of box-plots showing the observed spiking rates, population by population. In a comparable fashion to the previous benchmark, the obtained results closely resemble one another, and considerations similar to those made in the previous experiment apply.

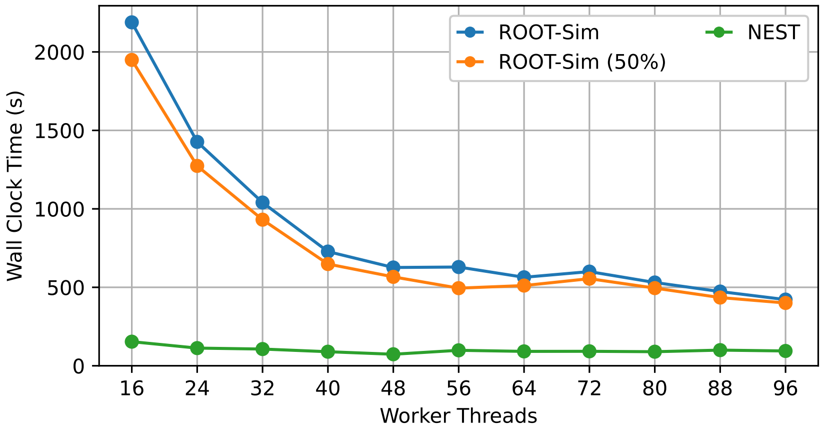
5.2 Performance Evaluation

Moving on, to assess the performance and scalability of our proposal, we have relied on running the Local Cortical Microcircuit (see Section 1.2.3) and the larger CUBA (see Section 1.2.2) models for 10 seconds of simulated time, and measured our implementation against state-of-the-art NEST implementations.

The results have been obtained by relying on a set of virtual machines on Amazon Web Services. In particular, we have used various configurations of m5.4xlarge instances (equipped with 16 virtual cores), m5.8xlarge instances (equipped with 32 virtual cores), and m5.24xlarge instances (equipped with 96 virtual cores). We compare our results with NEST, from a performance and scalability perspective. All results are averaged over 5 different runs. The same sequence of random numbers has been used across all runs in the experiment.



(a) CUBA.



(b) Potjans and Diesmann's.

Figure 5.3. Single-node Scalability (m5.24xlarge).

To evaluate the ability of the simulators to leverage multi-threading on a single machine, we executed the experiments using a 96-vCPU m5.24xlarge instance. In Figure 5.3 we report the total execution time (in seconds) for both benchmarks when varying the number of threads. We have also studied both simulators' performance when varying the total amount of interconnections in the network, i.e. running from the full load (as in the original benchmark) down to 25% of the total number of connections among neurons.

In both scenarios, we see that NEST's performance is mostly independent of the number of threads used and the number of considered connections[1]. This is related

[1] For the sake of readability, we report only the curve related to the full connection configuration for NEST, but the results with a reduced connection density showed the exact same performance.

to the approximated time-stepped nature of the simulation algorithm. Indeed, most of the time is spent by the NEST kernel advancing through the different steps and synchronising the various threads (through OpenMP [61]).

Conversely, our implementation exhibits a performance improvement that grows with the number of parallel threads used. At maximum parallelism, our simulations deliver a performance improvement of 4x or more in both benchmarks. More interestingly, the scalability trend shows that the minimum in the simulation time curve has not been reached yet, suggesting that the amount of parallelism that the Time Warp simulation can exploit is still non-minimal.

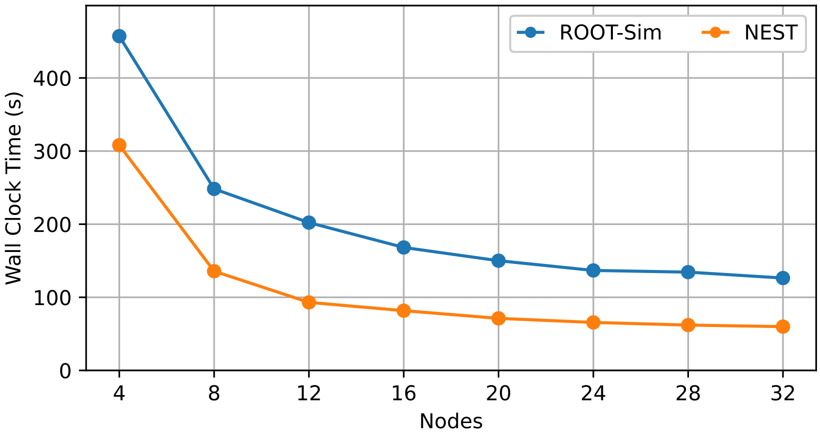
While the constant execution time of NEST consistently outperforms our implementation in the case of a smaller network (Potjans and Diesmann's model in Figure 5.3b), in the case of a larger network, NEST's performance starts to degrade at around 56 threads, when our solution starts to outperform it. This is an indication that, if larger networks were to be simulated, our solution could provide reduced simulation times—again, also offering results that are precise. This is a significant result, as the research community has the ambitious goal of "simulating the brains of mammals with a high level of biological accuracy and, ultimately, to study the steps involved in the emergence of biological intelligence" [38].

We have also studied the performance and scalability of our proposal, again against NEST, in a distributed environment, relying on MPI for both simulators. We have performed a strong scalability assessment, keeping fixed the size of the networks simulated in both benchmarks and varying the number of nodes from 4 to 32. We have used a set of m5.4xlarge instances (Figure 5.4) and a set of m5.8xlarge instances (Figure 5.5). These virtual machines are equipped with 16 and 32 virtual cores, respectively—we thus have used up to 512 distributed virtual cores, mimicking a tiny-scale supercomputer. By the results in Figure 5.3, this is a worst-case scenario for our implementation, as NEST is still outperforming ROOT-Sim on a single node in both configurations for both benchmarks.

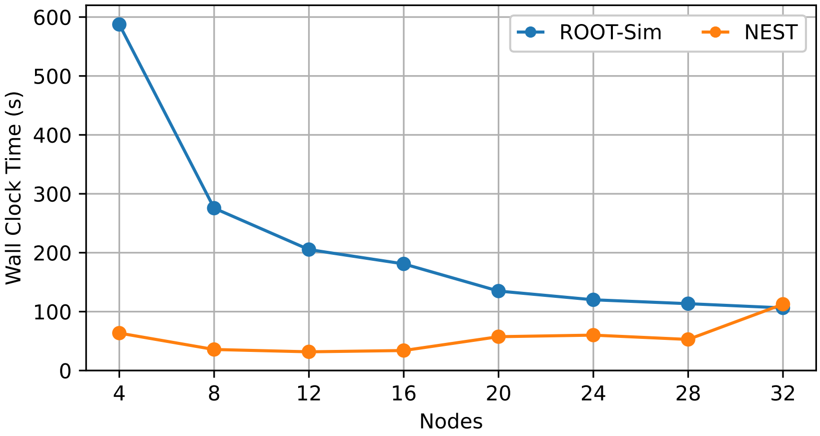
From the results, we anyhow observe that the scalability trends of both ROOT-Sim and NEST are comparable for the CUBA benchmark (Figures 5.4a and 5.5a). This is an indication that our proposal (particularly owing to retractable messages and publish/subscribe events) can dampen out the performance penalty observed on the single node in the case of a distributed simulation. Conversely, in the case of a smaller network (Figures 5.4b and 5.5b) the NEST implementation does not scale, probably due to synchronisation overhead, although, in most of the configurations, it is still able to deliver better performance results—again, sacrificing preciseness.

To better understand the dynamics of our PDES speculative simulation with respect to NEST's time-stepped simulation, we have also carried out an experiment using the large network in CUBA benchmark when varying the network activity, i.e. the total number of spikes in the simulation. From the results reported in Figure 5.6, we see that NEST exhibits a performance that is independent of the amount of activity in the network (Figure 5.6b), while the Time Warp simulation based on ROOT-Sim can exploit this reduced amount of interactions. This is an expected result, given the nature of both simulation methodologies. At the same time, it is interesting to note that when the number of nodes is reduced, the benefit incurred from reducing the network activity is more noticeable (see Figure 5.6a). This is related to the fact that more LPs are bound to the same thread when the number of nodes is smaller. In this scenario, the publish/subscribe messaging mechanism that we have devised is likely to pay off more at higher node counts. It follows that if larger networks were to be executed, the performance improvement offered by this runtime support can be non-minimal.

Finally, to shed light on the reason behind the performance gap between the ROOT-Sim implementation and the NEST one (although we recall that the results have been obtained in a worst-case scenario), we report in Figure 5.7 a breakdown of the time spent in the initialisation phase vs the simulation phase for the CUBA benchmark on the cluster of m5.8xlarge virtual machines. As it can be seen, the simulation pays a non-negligible almost-constant time spent in the initialisation phase, while the simulation phase is scaling significantly. This indicates that if more effective initialisation strategies are devised, our approach might also improve performance when simulating smaller-scale networks.

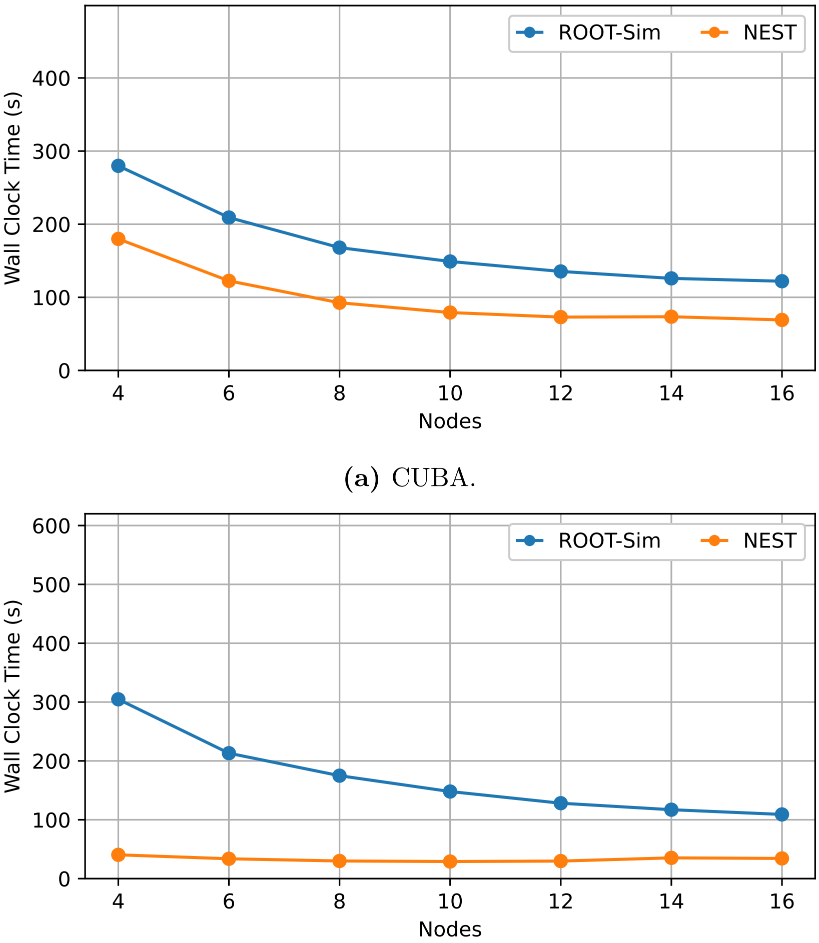


(a) CUBA.



(b) Potjans and Diesmann's.

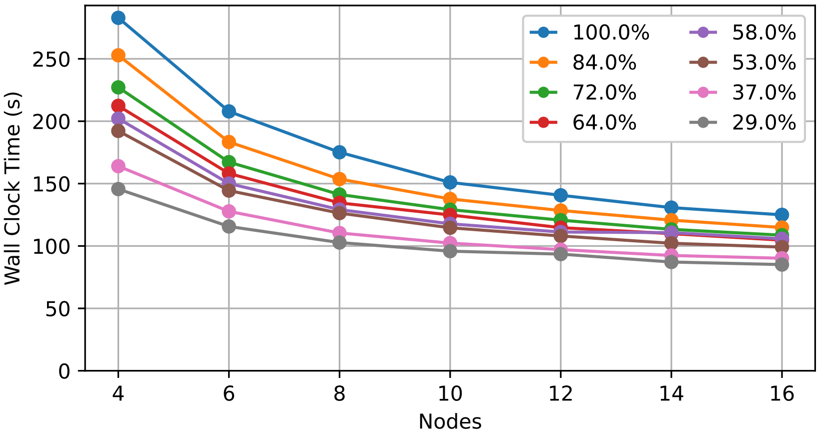
Figure 5.4. Distributed Scalability (m5.4xlarge).



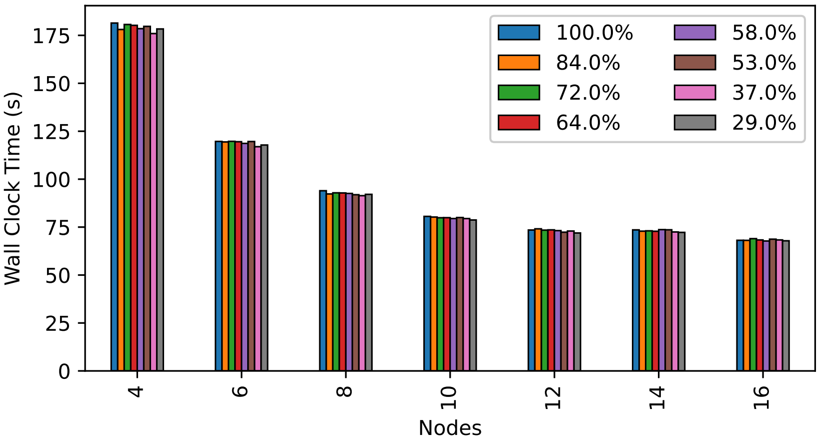
(a) CUBA.

(b) Potjans and Diesmann's.

Figure 5.5. Distributed Scalability (m5.8xlarge).



(a) ROOT-Sim.



(b) NEST.

Figure 5.6. Performance when varying network activity (m5.8xlarge).

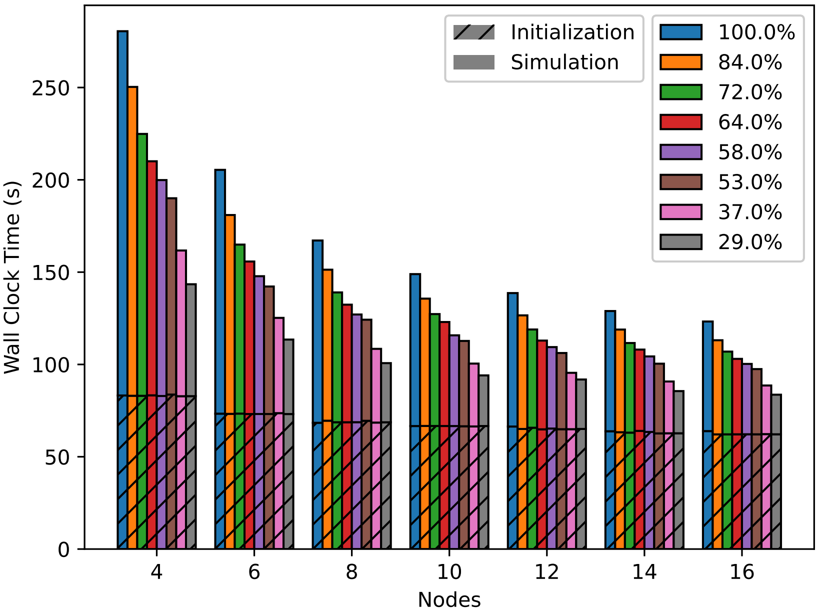


Figure 5.7. Initialisation vs simulation time (m5.8xlarge).

5.3 Accuracy Evaluation

After evaluating the accuracy at an ensemble level, by comparing frequencies, we dove deeper to explore our accuracy claims at a neuron-grained level. By comparing the spike times of the single neurons, rather than the spike frequencies of populations, we managed to gather data that paint an interesting picture.

2 neurons. We started with the tiny 2-neuron network (see Section 1.2.2), that was deemed the ideal model to commence the evaluation of the approach, with such fine grained precision. In the experiment, the spiking outputs of neuron N_2 are monitored, and then compared with a ground truth. In particular, to generate the ground truth, we have computed the timing of N_2's expected spike train via numerical methods with a tolerance of 10^{-7} milliseconds. The computed ground truth has been used as a reference for the result obtained simulating the model both on NEST and ROOT-Sim. We have collected data for 1 second of simulated time.

In Figure 5.8 we report the accuracy results. In the Figure, we plotted the absolute error for each i^{th} spike in the train. While the accumulated error grows linearly for both the simulation methods, it is clear that the error in the NEST

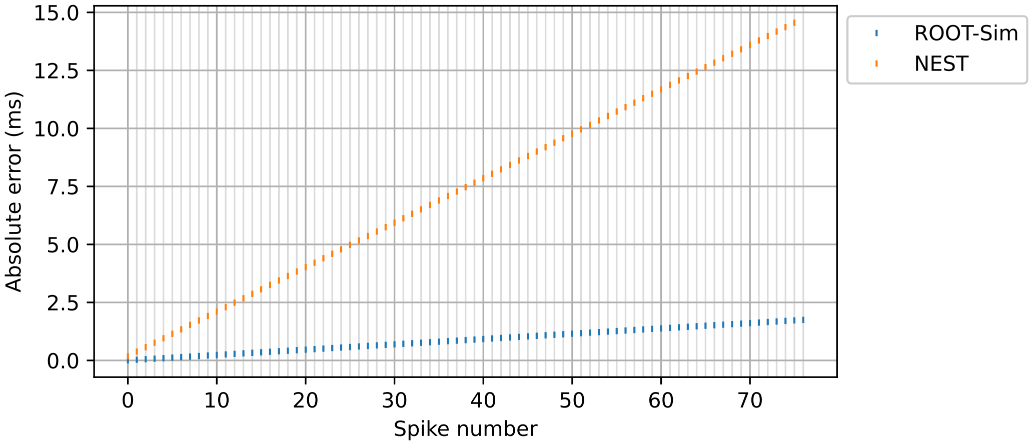


Figure 5.8. Accuracy results with a 2-neuron model.

model is significantly higher, coming up to a total of 15 ms of error over 1 second of simulated neuronal activity (ROOT-Sim shows an error reduction of $\sim$ 90% compared to NEST). Again, this is the case with an extremely simple network with two neurons, and in a very short time window. This error is also reflected in the number of spikes—NEST runs late and thus misses one spike, with respect to the total number of spikes in the considered simulation window.

Feed-Forward Precision Benchmark. In order to further assess the accuracy capabilities of the different approaches, we have used the Feed-forward Precision Benchmark introduced in Section 1.2.2.

The evaluation method consists in comparing the spike timings of the *Output* population against the expected ones. We evaluate both NEST and ROOT-Sim.

First of all, we need to establish a method to compute the expected behaviour, the ground truth. As noted when presenting the model, the network chosen for the accuracy evaluation is acyclic, and, conveniently, there is a simple algorithm able to compute its behaviour. Given such an acyclic network, we compute a topological ordering of the neurons $n_0, n_1, ...n_k$; then, necessarily, the behaviour of a neuron n_i will only depend on the behaviour of neurons $n_0, n_1, ..., n_{i-1}$. It follows that, once a simulation time limit t has been selected, it is possible to simulate the neurons one by one, starting from n_0 through n_k feeding the output from each of the neurons to the correct post-synaptic ones. Since there is no analytical closed-form solution for

Table 5.1. Spiking Times for ROOT-Sim and NEST (tolerance/time-step: 0.1 *ms*). For each result, we provide the spike time (*ms*) and the neuron number in brackets. The results relate to the first 10 *ms* of simulated time.

Spike No.	Ground Truth	ROOT-Sim	NEST
1	2.999 (900)	3.046 (900)	3.200 (900)
2	3.556 (977)	3.615 (977)	3.900 (975)
3	3.598 (975)	3.630 (975)	3.900 (950)
4	3.787 (970)	3.771 (970)	6.200 (912)
5	5.843 (953)	5.955 (953)	6.300 (952)
6	—	6.215 (927)	—
7	—	6.667 (923)	—

Table 5.2. Spiking Times for ROOT-Sim and NEST (tolerance/time-step: 0.001 *ms*). For each result, we provide the spike time (*ms*) and the neuron number in brackets. The results relate to the first 10 *ms* of simulated time.

Spike No.	Ground Truth	ROOT-Sim	NEST
1	2.999 (900)	2.999 (900)	3.110 (900)
2	3.556 (977)	3.556 (977)	3.795 (975)
3	3.598 (975)	3.598 (975)	6.486 (952)
4	3.787 (970)	3.787 (970)	—
5	5.843 (953)	5.842 (953)	—

the spike times for the LIF neuron used in the network, we still have to resort to numerical methods. However, we are not concerned with performance in this case, just accuracy. Therefore, to compute the ground truth, we implemented a Python script capable of carrying out the described computations. The script finds spike timings using bisection, stopping after the diameter of the search interval (i.e. the tolerance) is less than 10^{-9} *ms*.

We report in Tables 5.1 and 5.2 the results obtained running the synthetic model on ROOT-Sim and NEST, compared to the ground truth results obtained according to the method described above. To compute the results, the simulations were run by setting NEST's simulation time-step and ROOT-Sim's bisection tolerance (i.e. the stopping size of the search interval's diameter, see Section 3.4.1) to 0.1 *ms* (results in Table 5.1) and 0.001 *ms* (results in Table 5.2). The results report the spikes emitted in the simulation's first 10 *ms*. We provide the spiking time and the ID of the neuron that generated the spike in the Output layer for each spike.

By the results in Table 5.1, we observe that, for both simulators, the accuracy is not high. In particular, ROOT-Sim generates spikes at all the correct neurons, but the difference in spiking times is between 1% and 2%, even in such a significantly reduced simulation time. Interestingly, this approach also generates two additional spurious spikes. Conversely, NEST has a higher error (up to 60%), but more interestingly, it induces spikes at the wrong neurons, except for the first one. The number of spikes is anyhow correct. These results are expected. Indeed, given the nature of the synthetic model, it is clear that a low resolution is unlikely to provide accurate results due to the strong interaction between excitatory and inhibitory neurons.

The results with a higher resolution, provided in Table 5.2, show that the results based on ROOT-Sim deliver much higher accuracy. Conversely, NEST results show that two spikes are missing, spikes are induced at the wrong neurons, and the accuracy is still low (with an error ranging from 3.7% to 80%). One could wonder how the two examined simulators may deliver a different accuracy even when using the same value for the time-step/error. We believe that the sources of inaccuracy are essentially two.

Common to both algorithms, the first source of inaccuracy comes from the computational inaccuracy when calculating spike timings: even small deviations can cause post-synaptic neurons to incorrectly emit or miss a spike. This source however, can be overcome by increasing the simulation resolution. The impact of increased simulation resolution will be evaluated in Section 5.3.1.

The second source of accuracy loss is specific to NEST only, and it depends on the way spike detection is implemented. With the default settings, a spike is detected only if the firing threshold potential is overcome at one discrete time-step. In other words, NEST notices and correctly emits a spike, only if the conditions are valid at the time of the time-step. The effect this has on accuracy is twofold: firstly, it can lead to missing a spike in the edge case in which the threshold is overcome only for a short period of time in-between time-steps, but the potential is below threshold at the time-step end; secondly, the spike timings are snapped to the time-step grid. This delays the spike timing to the end of the time-step, even when the membrane potential would have surpassed the spiking threshold earlier. In the worst case, the threshold could be surpassed immediately after the start of the time-step.

These inaccuracies have a compounding effect that is well visible already in this simple experiment, let alone in larger and more convoluted networks.

5.3.1 The Trade-off Between Accuracy and Performance

To conclude the experimentation regarding accuracy, we have conducted a series of performance experiments. The performance experiments were run on an AWS m5.8xlarge virtual machine with 32 vCPUs. These machines are based on Intel Xeon Platinum 8175M processors, running Ubuntu 20.04.3 LTS, on kernel version 5.13.0-1025-aws. Each experiment was run with 32, 24, 16, 8, and 4 worker threads. NEST only has data points for 16 or more worker threads due to a limitation not allowing more than 2^{27} synaptic connections per worker thread. While it seemed possible to change this number in the code, after some research, we ultimately decided not to venture into that, to avoid causing any unwanted issues with the simulator. Because of this, only ROOT-Sim was run on 4 and 8 workers.

Tested using the CUBA model (see Section 1.2.2) with 300,000 neurons, the model is simulated for 10 seconds of simulation time with each simulator, while varying the simulation precision. In NEST, this is achieved by selecting a resolution value. In our implementation, the time tolerance is built into the model at compile time and can be selected deliberately, as long as the hardware constraints allow it.

Figure 5.9, shows the results of the performance experiments. While NEST outperforms ROOT-Sim in terms of speed for low-resolution values (10^{-1} and 10^{-2} milliseconds), when running with a resolution of 10^{-3}, the performance dramatically degrades, leaving the edge to ROOT-Sim, even when the latter runs on eight workers. With a resolution of 10^{-4}, the time-to-solution for NEST is significantly larger, with the best configuration (using 32 worker threads), taking over $20,369$ seconds to complete, while ROOT-Sim took $1,076$—that is, NEST takes 18,92x the time. This result is expected, as multiplying the resolution tenfold also multiplies the number of calculations needed. It is not unreasonable to expect higher resolutions to be practically unfeasible for sizeable networks.

At the same time, increasing simulation resolution appears to have a minimal

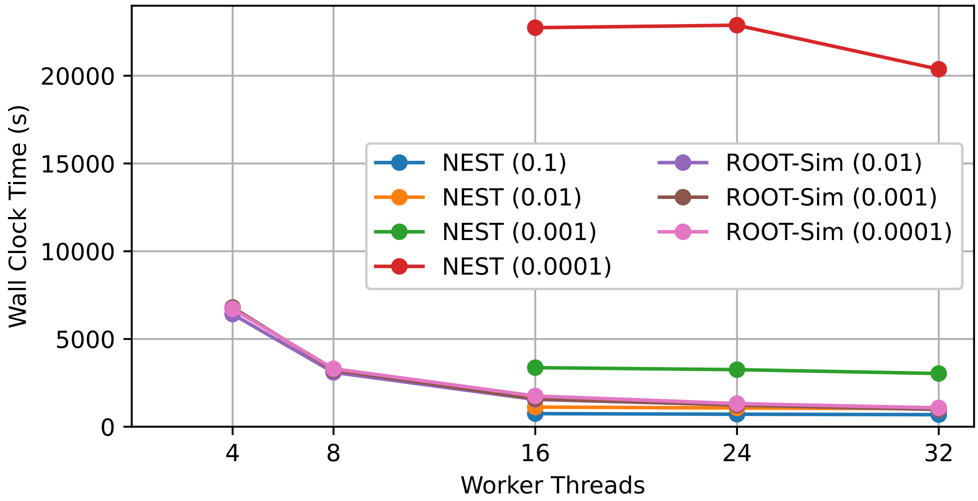


Figure 5.9. Performance Comparison.

impact on ROOT-Sim's performance, allowing it to be increased almost at will without the risk of running into prohibitive time costs, the only limitation being the implementation's floating point precision.

5.4 Braille Letter Reading

Starting from the high accuracy the simulation method is able to deliver, we want to evaluate the effect of the increased accuracy when executing AI workloads. For this reason, we re-implemented the network presented in [44] in ROOT-Sim.

The cited paper proposes a network implementation to perform braille letter recognition. Braille is a tactile writing system for blind and visually impaired people. The letters are represented by using raised bumps on a grid, arranged in three rows of two dots each. The combination of the six raised/flat dots determines the letter represented.

The authors suggest that braille reading, as a spatio-temporal pattern recognition task, can be an interesting test-bench for robotics applications. To perform pattern recognition, they use a SNN that they also then deploy on the Intel Loihi digital neuromorphic chip for fast and efficient inference. Loihi uses Current Based LIF neurons and first-order integration to calculate the evolution of neural dynamics.

For the network, they propose two topologies, which are effectively different ways to connect the same sets of neurons. The neurons are divided in three sets: input layer of $k * 24$ neurons, a hidden layer of 450 neurons, and the output layer of 27 neurons, one per letter, plus the space character. The first topology is a Feed-Forward SNN (FFSNN), in which the input layer is fully connected to the hidden layer, and the hidden layer is fully connected to the output layer. The second topology is a Recurrent SNN (RSNN), which is the same as the FFSNN, but with each layer also recurrently fully connected to itself.

To perform the sensing, the authors use a robotic fingertip with 12 sensors. Each of the sensors emits a continuous signal, with varying intensity. The produced signal can have positive and negative values. To emulate *event-driven sensors*, the authors "spikify" the continuous signal generated by the sensors into two possible discrete signals, ON and OFF. Using a sampling threshold θ, the ON spike is generated whenever the signal intensity increases and crosses a multiple of θ, while the OFF spike is generated when the signal intensity decreases, and crosses a multiple of θ. E.g. a signal going from 0 to 11.2 and then to 7, will generate 11 ON spikes followed by 3 OFF spikes with $\theta := 1$. With $\theta := 2$, the signal will be converted into 5 ON spikes (when reaching 2, 4, 6, 8, 10) and 2 OFF spikes (when reaching 10, 8), and so on. The datasets are generated for values of $\theta \in \{1, 2, 5, 10\}$. Increasing θ decreases the number of spikes and thus the sensitivity of the spikification.

The spikification of the dataset is the reason for having 24 input neurons: each of the 12 fingertip sensors can generate 2 signals, thus each of the neurons in the input layer is responsible for generating either an ON or OFF spike, when needed. The multiplying factor k is used to increase the number of input copies: for higher values of θ, the input signal can become very rarefied, with spikes distant from one another. Thus, instead of a single spike, k spikes are sent per ON/OFF signal.

To simulate and train the network, the authors use a time-stepped approach based on PyTorch. Consequently, there is a limit to the number of spikes an input neuron can generate: a neuron cannot spike twice in the same time-step. For this reason, the spikified dataset is further adapted by making the input neuron generate a spike at time-step t if in the time window $t + \Delta t$ at least one spike is emitted, thus

Table 5.3. Reported Best Hyperparameter Configurations.

θ	k	Δt (ms)	τ_{syn} (ms)	τ_{mem} (ms)
1	2	3	6	60
2	8	5	5	50
5	4	5	7	70
10	2	3	7	70

ignoring all further spikes for the time-step. Needless to say, such approximation would not be needed when simulating with PDES (or using an analogue neuromorphic chip): since the input neurons are simply there to model spike trains, then they do not have a limit to their spiking rate, meaning all spikes would be delivered to the hidden layer's neurons at the appropriate time, which would have avoided having to squash the sensor dynamics in this way.

The network's prediction is obtained by counting the spikes of the output neurons: the character associated with the neuron that spiked the most is the chosen classification for the input signal.

A vast exploration phase for hyperparameter optimisation is conducted in the paper. For the sake of brevity, we focus only on the hyperparameter configurations yielding the best performance, according to the authors. Table 5.3 reports the hyperparameters chosen for each of the values of θ.

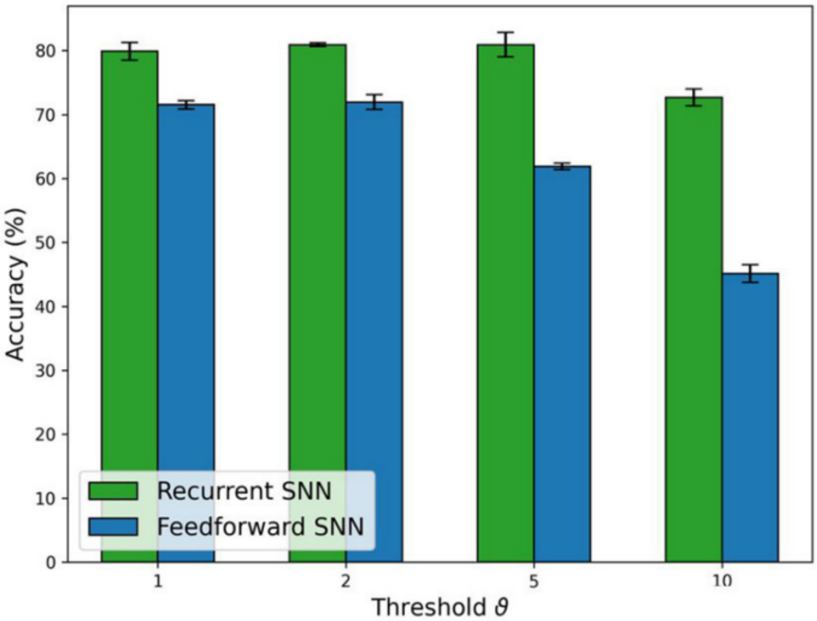


Figure 5.10. Reported Classification Accuracy, [44] Figure 7

Figure 5.10 shows the classification accuracies that the paper reports as being achieved by the networks, when trained using the best hyperparameters.

Moving on to our experimentation, we executed the PyTorch training script provided by the authors using the optimal hyperparameters, which generates 5 sets of weights, extracted the trained weights, and used them in the re-implemented network model we built in ROOT-Sim. The accuracies we reproduced after training and simulating using the PyTorch script are reported in Figure 5.11. As one can see, there is an issue with the reproduced accuracy for RSNN at $\theta := 10$, as it plummets for no apparent reason, while the rest of the graph is comparable to the paper's. After multiple tries, it became apparent that this stems from an issue in the original authors' training script, which we did not focus on solving. The case of $\theta = 10$ will thus not be treated for RSNN. We now explain the process of porting the simulation to ROOT-Sim, which proved to be an interesting task.

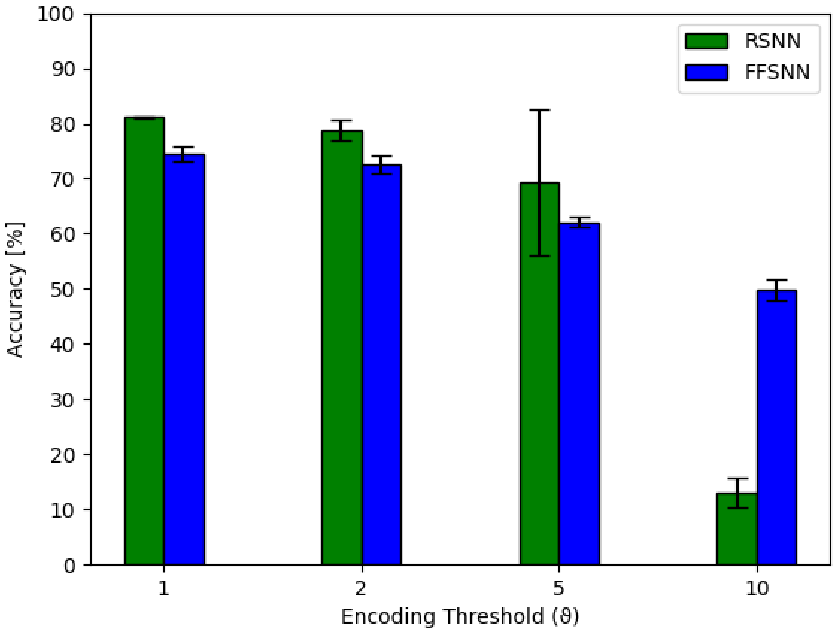


Figure 5.11. Reproduced Accuracy, reproducing Figure 7 of [44]

5.4.1 Taming the RSNN

It turns out that the time-stepped implementation of the network using PyTorch had no explicit transmission delays, nor refractory periods: the large values used for Δt are implicitly defining these characteristics, as we have seen in the accuracy evaluation presented in Section 5.3. Indeed, spikes are aligned to the grid, and at most one spike can be generated by a neuron per time-step Δt, limiting the neuron's spiking capabilities. Furthermore, spikes generated are (in this implementation) delivered at the start of the next time-step, resulting in an implicit spike transmission delay of Δt.

As a result, the RSNN model ported to ROOT-Sim would spike uncontrollably, thrashing the simulation. This happened because of recurrent connections: without a refractory period, all it took was few neurons to spike in close succession to one another to kick-start a positive feedback loop that generated an exploding number of spikes. For this reason, we inserted both a refractory period τ_{ref}, and a forced spike generation delay d_f, while the spike propagation delay was set to the Δt used in the time-stepped version of the same network.

While unorthodox, the role of d_f was to simulate the implicit spiking delay introduced by the time-stepping with large Δt values. Differently from what happens during a refractory period, in which a neuron ignores all incoming spikes, the forced delay just enforced spacing between consecutive spiking of the same neuron. As such, a neuron's membrane potential would be able to rise, but no spike would be produced before d_f had passed since the last time the neuron fired.

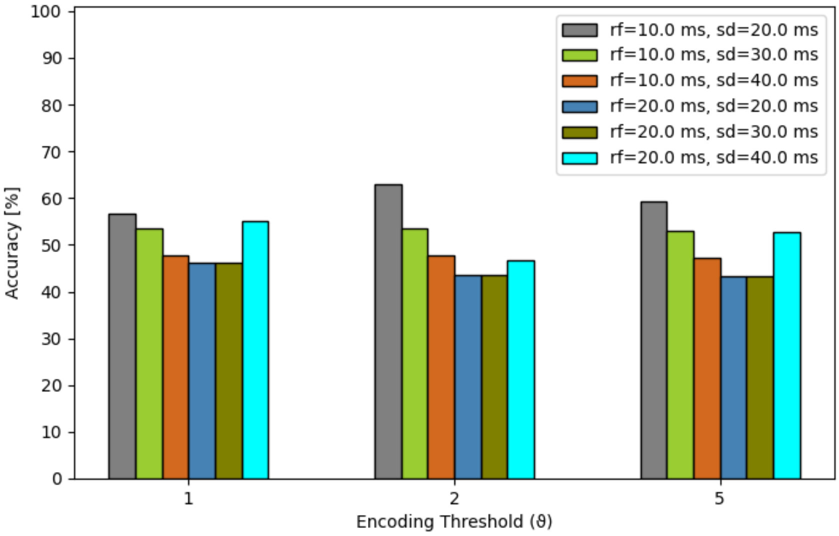


Figure 5.12. Sweeping on τ_{ref} and d_f. rf: τ_{ref}, sd: d_f

To pick the correct values for τ_{ref} and d_f, we executed a parameter sweep, producing one experiment per combination in the Cartesian product of a number of arbitrary values for $\tau_{ref} \in \{0, 2.5, 4, 5, 7.5, 10, 20\}$ and the (again, arbitrary) values for $d_f \in \{0, 20, 30, 40\}$. This was done once per value of $\theta \in \{1, 2, 5\}$. Figure 5.12 reports the plot showing, among others, the best configuration found with this parameter sweep: $\tau_{ref} = 10ms$ and $d_f = 20ms$.

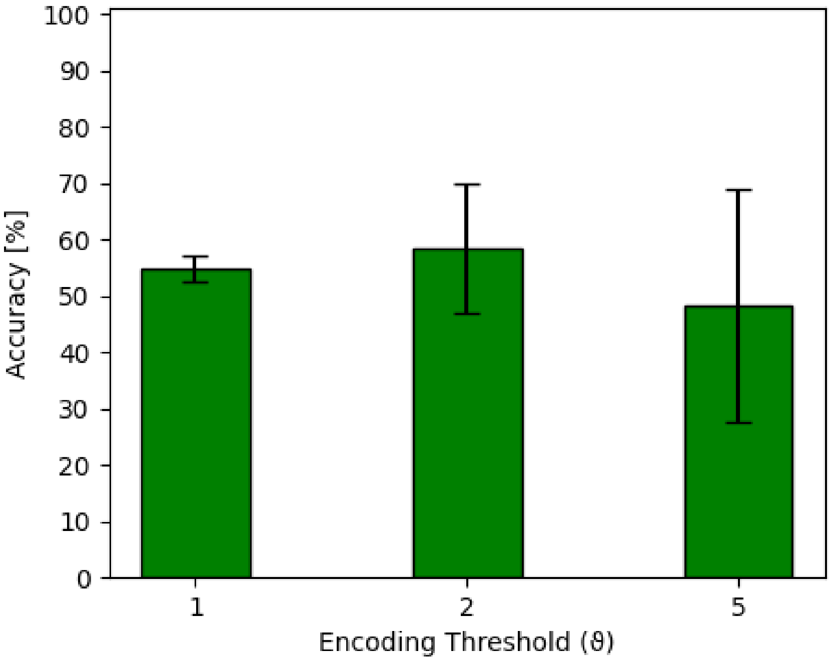


Figure 5.13. Box-plots of accuracies for $\tau_{ref} = 10ms$, $d_f = 20ms$

The resulting box-plot of prediction accuracies over the 5 different weight sets, is shown in Figure 5.13. We can see how the performance is not really comparable with the one presented in the original paper, nor with the one we reproduced. By scaling down the weights of the recurrent connections and performing a new parameter sweep, a new configuration was found with better accuracy, shown in Figure 5.14.

However, this configuration is still slightly lacking with respect to both the presented configuration and the learned one. We have to keep in mind that the weights were learned on a network with a significantly different behaviour: the achieved results are already vastly superior to what we started with, thanks to the parameter sweep. This is also a prime occasion to bring attention to the fact that, if deployed on an analogue chip, this network would have behaved in a way similar to ROOT-Sim, showing once again how, since the accuracy losses make the simulated behaviour drift away from that of the real implementation, renders the simulation without accuracy much less reliable and useful for prototyping networks deployed on analogue chips.

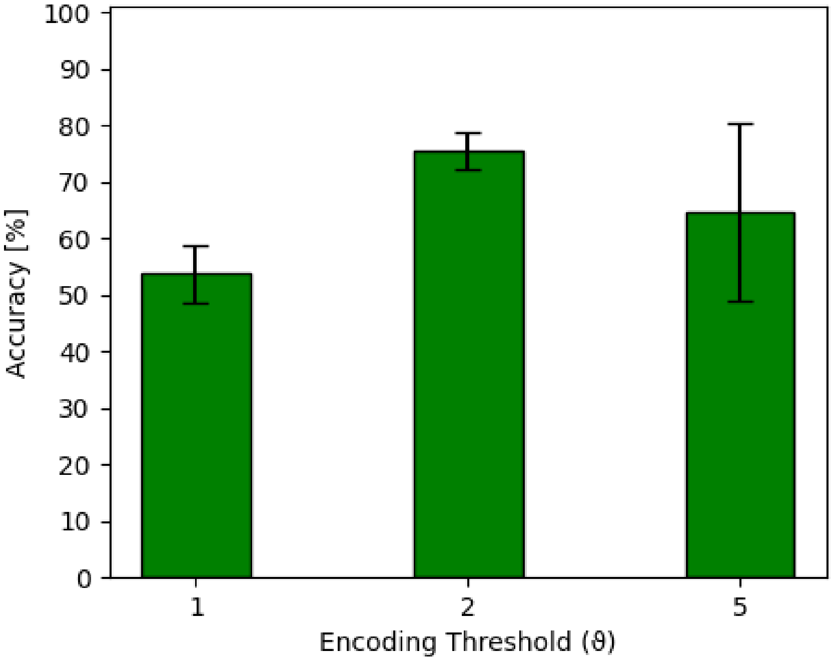


Figure 5.14. Accuracies with scaled-down recurrent weights, $\tau_{ref} = 0.25ms$, $d_f = 0ms$

5.4.2 FFSNN Takes Over

Porting and simulating the FFSNN was a much more straightforward process, with respect to the RSNN: the absence of recurrent connections makes an explosive positive feedback loop impossible. As far as the other aspects are concerned however, the network still carries the same quirks as the RSNN: implicit refractory period and spike propagation delay determined by the simulation time-step Δt. However, to solve the issues in the case of the FFSNN, all that was needed was a parameter sweep to find an appropriate refractory period τ_{ref}, while setting the synaptic propagation delay to the Δt used in the time-stepped version of the network. The results of the parameter sweep are shown in Figure 5.15.

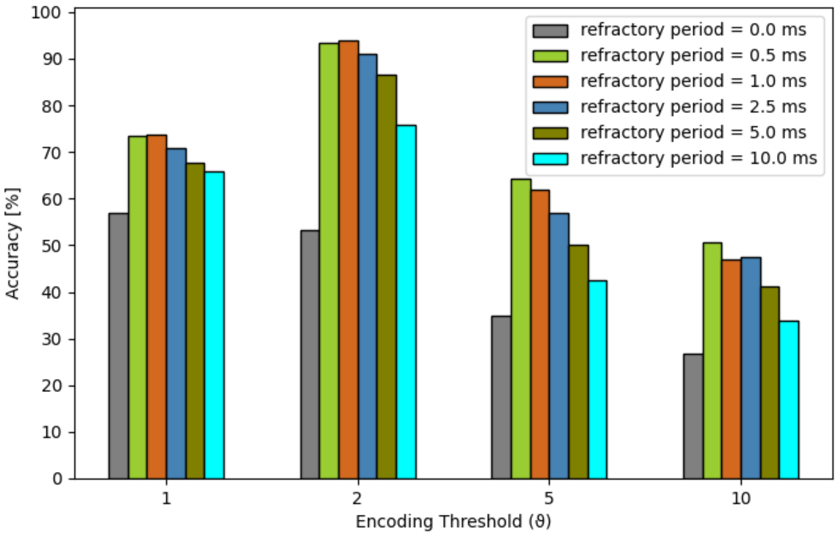


Figure 5.15. Sweep over τ_{ref} for FFSNN.

For FFSNN we also find $\theta = 10$ as a possible encoding threshold, as it did not cause issues with FFSNN. The best value for τ_{ref} was found to be $1.0ms$. Then, with the found value, the prediction accuracies were gathered by simulating using the 5 different weight sets.

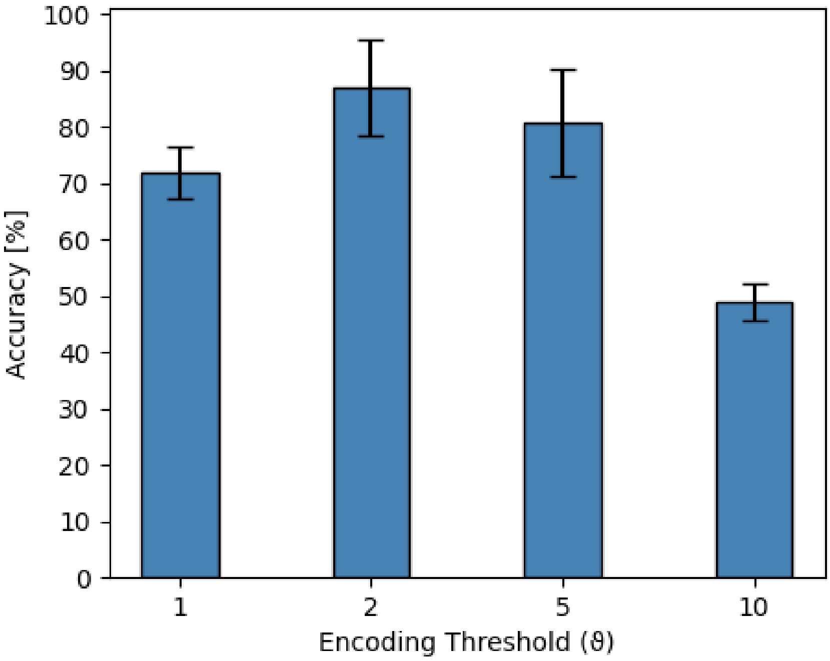


Figure 5.16. FFSNN accuracies with $\tau_{ref} = 1.0ms$

Figure 5.16 shows the resulting accuracies for FFSNN. In a surprising (or maybe at this point, not so much) turn of events, this configuration greatly outperformed all accuracies, both reported in the paper, and reproduced, even when comparing to RSNN. This FFSNN exceeds expectations at every single value of θ. The most obvious difference is found for $\theta = 2$, which shows a peak improvement of over 20% with respect to the time-stepped FFSNN, and up to 10% with respect to the RSNN. It is impressive that this result was achieved simply with a parameter sweep over the refractory period, and by using a more accurate simulation of the network.

5.4.3 Energy Efficiency

After appreciating the capabilities brought about by a more accurate simulation, it is time to use the gathered data to evaluate the energy efficiency that the hardware-implemented network would have. We compare the RSNN, as the best performing network reported in the paper, and the best performing FFSNN executed using ROOT-Sim.

To estimate the electrical consumption, the metrics reported in [47] have been used. These are based on existing dedicated hardware's energy consumption. To compute the energy consumption, the number of spikes emitted by each neuron is collected.

According to gathered data, for a complete classification run, the hidden layer of the RSNN on average emits around 10^6 total spikes, while the FFSNN's hidden layer simulated on ROOT-Sim emits around 10^7 spikes, on average. Taking into account the size of the hidden layer (450 neurons), and the size of the used test-set (1085 elements), this means that on average the neurons of the RSNN's hidden layer emit 2 spikes each, while the neurons in the FFSNN's hidden layer emit, on average, 20 spikes each.

For the power consumption estimation, we use Equation 9 from [47]:

$$E_{syntot} = E_{syn} + \frac{E_{neu}}{r_a \cdot s_{neu}} \tag{5.1}$$

Where:

E_{syntot} = Total energy dissipated for a message to pass through a synapse.

$$E_{syn} = \text{Dissipated energy per synapse}$$

$$E_{neu} = \text{Energy dissipated by a neuron}$$

$$s_{neu} = \text{average number of synapses per neuron}$$

$$r_a = \text{percentage of active synapses}$$

We observe that $r_a \cdot s_{neu}$, is the number of synapses on which the spike has been propagated. Since the topology is complete, $r_a = 1$. In the RSNN $s_{neu} = 450 + 27$ (recurrent connections + output neurons), while in the FFSNN $s_{neu} = 27$, because no recurrent connections are in place. Output neurons are counted as having a single output synapse (i.e. $s_{neu} = 1$ for them). Since Equation (5.1) calculates the energy consumption of a single synaptic event, to have the energy consumed per spike generated, we just multiply by s_{neu}, obtaining:

$$E_{spike} = s_{neu} \cdot E_{syntot} = (s_{neu}) \cdot (E_{syn} + \frac{E_{neu}}{1 \cdot s_{neu}}) = s_{neu} \cdot E_{syn} + E_{neu} \tag{5.2}$$

Using Equation (5.2), we can compute the energy consumed. Using average consumption values $E_{syn} = 621.645aJ$ and $E_{neu} = 2477.112aJ$, we obtain the results shown in Figure 5.17.

As we can see, using this simulation method we have been able to estimate that not only will the FFSNN perform better in terms of accuracy, but it will even consume a noticeably smaller amount of energy, in the case of analogue implementations. This made possible thanks to the increased accuracy provided by our method, which allowed more precise simulation of the neurons and of the chip's behaviour, leading to improved accuracy with a simpler network, which in turn resulted in a lower estimated energy consumption.

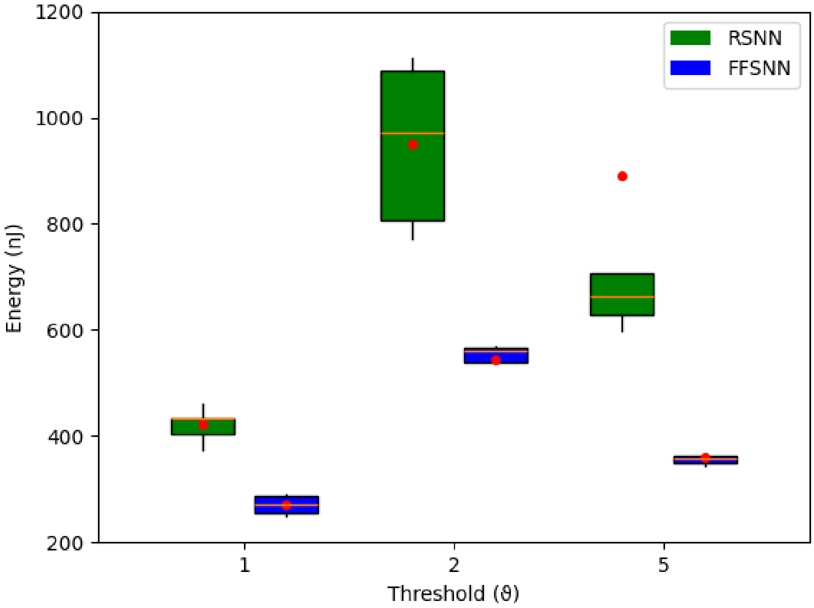


Figure 5.17. Estimated energy consumption for executing the different networks on neuromorphic hardware.

Chapter 6

Conclusions and Future Work

In this thesis, we have presented an effective framework to carry out the simulation of Spiking Neural Networks using Parallel Discrete Event Simulation. To do so, we had to overcome innumerable obstacles in what at the start was—and because of its sheer complexity still somehow is—a fascinating but daunting field. From having to come up with a way to simulate a continuous-time system using discrete events, to understanding and translating the models into workable materials, to debugging elusive concurrency issues, the path to produce the presented research was deeply challenging, and the time not nearly enough (although is the time ever enough?).

As developers and engineers at heart, we have indeed managed to model a continuous neuron by only using the salient instants of its evolution, and we managed to do so with better scalability and accuracy than what was possible before. To support a scalable and fast execution, we developed complex runtime supports the practical reach of which goes well beyond the scope of SNN simulation: retractable events and publish/subscribe message dissemination. To help others reap the benefits of this innovative SNN simulation approach, we encapsulated the PDES backend with the neural simulation interface that hides away its complexities. Then we encapsulated once more, by wrapping Python bindings around the models implemented using the neural simulation interface, to further lower the barrier to adoption and to make model development and deployment faster and easier.

We then demonstrated how small inaccuracies compound and make the simulation drift from reality, and we underlined how accurate simulation of Spiking Neural

Networks is vital for the future development of the field, by providing working examples of the efficiency and efficacy of accurate models.

All of this with the hope that this will prove useful to further the research in AI, medicine, biology and much more. With this research, we believe to have pushed the boundaries of what is possible a tiny step forward.

We conclude this thesis, by proposing future developments for this line of research that hold promise in the eyes of the author.

Models Without Analytical Solution. The natural continuation of the endeavours presented in this thesis, passes through the implementation of more complex neuron models, most of which have no analytical solution. While the absence of an analytical solution prevents part of the reasoning we used to make the future-spike-timing calculation faster, and imposes the need for some restructuring on the approach to neuronal state evolution, the PDES approach still has the capacity to offer advantages with respect to time-stepping.

Indeed it still holds that PDES does not need a snapping to grid of neuronal dynamics nor spike timings. Thus, while the model evolution will have to be performed with iterative numerical methods, when using PDES there is no constraint on the step size to be used. As such, a desired precision level can be selected, and more complex numerical methods employing dynamically sized integration step—such as the Runge-Kutta-Fehlberg [17] method, widely used in other scientific and engineering fields—can be used to ensure that a high level of accuracy is guaranteed, while at the same time preventing excessively complicated calculations from having to be done every single integration step for the sake of maintaining accuracy.

Sparse Coding by Spiking Neural Networks. At the time of writing of this thesis, the author is working on reproducing the work presented in [66]. In the mentioned work, the authors present a technique to perform sparse coding using SNNs. Their adopted approach is time-stepped, as their focus is to demonstrate the functionality of the algorithm, and the possibility to deploy it on the Intel® Loihi® neuromorphic chip, which is a digital chip that uses first order numerical methods to implement time-stepping.

Similarly to what we presented in Section 5.4, the aim of this effort would be to demonstrate the centrality of achieving high accuracy in SNN simulation, in order to achieve higher usability and precision for algorithms implemented using simulated SNNs but also, and perhaps most importantly, to accurately model analogue hardware implementations of SNNs.

The experimentation we conducted and presented in this thesis has shown how, when using the classical time-stepped approach, the behaviour of a simulated SNN actually quickly diverges from that of the circuit it is supposed to be executing, due to the compounding effect of small inaccuracies. This prevents us from being able to adequately reproduce the physical network's behaviour, precluding the possibility to properly train it and rely on it.

The relevance of this further development lies in the fact that, while the research in Section 5.4 is executing an AI task and workload, in this case the network has special properties (far from biological ones) and is used to execute an algorithm (which, ironically, is vastly used in AI).

Who knows what the future holds.

Bibliography

[1] Larry F Abbott. 1999. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain research bulletin* 50 (1999), 303–304. `https://doi.org/10.1016/S0361-9230(99)00161-6`

[2] Rajagopal Ananthanarayanan, Steven K Esser, Horst D Simon, and Dharmendra S Modha. 2009. The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*. ACM, New York, NY, USA, 1–12. `https://doi.org/10.1145/1654059.1654124`

[3] Peter D Barnes, Christopher D Carothers, David R Jefferson, and Justin M LaPre. 2013. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS '13)*. ACM, New York, NY, USA, 327–336. `https://doi.org/10.1145/2486092.2486134`

[4] Mark W Barnett and Philip M Larkman. 2007. The action potential. *Pract. Neurol.* 7, 3 (June 2007), 192–197.

[5] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C Stewart, Daniel Rasmussen, Xuan Choo, Aaron Russell Voelker, and Chris Eliasmith. 2014. Nengo: a Python tool for building large-scale functional brain models. *Frontiers in neuroinformatics* 7 (2014), 13. `https://doi.org/10.3389/fninf.2013.00048`

[6] Luiz Severo Bem Junior, Nilson Batista Lemos, Luís Felipe Gonçalves de Lima, Artêmio José Araruna Dias, Otávio da Cunha Ferreira Neto, Carlos Cezar Sousa

de Lira, Andrey Maia Silva Diniz, Nicollas Nunes Rabelo, Luciana Karla Viana Barroso, Marcelo Moraes Valença, and Hildo Rocha Cirne de Azevedo Filho. 2021. The anatomy of the brain - learned over the centuries. *Surg. Neurol. Int.* 12 (June 2021), 319.

[7] Mohammad A Bhuiyan, Vivek K Pallipuram, Melissa C Smith, Tarek Taha, and Rommel Jalasutram. 2010. Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–8. https://doi.org/10.1109/IPDPSW.2010.5470899

[8] T Binzegger. 2004. A Quantitative Map of the Circuit of Cat Primary Visual Cortex. *Journal of Neuroscience* 24 (2004), 8441–8453. https://doi.org/10.1523/JNEUROSCI.1400-04.2004

[9] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M Bower, Markus Diesmann, Abigail Morrison, Philip H Goodman, Frederick C Harris, Jr, Milind Zirpe, Thomas Natschläger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P Davison, Sami El Boustani, and Alain Destexhe. 2007. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience* 23, 3 (Dec. 2007), 349–398. https://doi.org/10.1007/s10827-007-0038-6

[10] Kristofor D Carlson, Michael Beyeler, Nikil Dutt, and Jeffrey L Krichmar. 2014. GPGPU accelerated simulation and parameter tuning for neuromorphic applications. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, Piscataway, NJ, USA, 570–577. https://doi.org/10.1109/ASPDAC.2014.6742952

[11] Nicholas T Carnevale and Michael L Hines. 2006. *The NEURON Book*. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9780511541612

[12] Andrew S Cassidy, Jun Sawada, Paul Merolla, John V Arthur, Rodrigo

Alvarez-Icaza, Filipp Akopyan, Bryan L Jackson, and Dharmendra S Modha. 2016. *TrueNorth: A High-Performance, Low-Power Neurosynaptic Processor for Multi-Sensory Perception, Action, and Cognition.* Technical Report. Almaden Research Center, IBM Research.

[13] Kit Cheung, Simon R Schultz, and Wayne Luk. 2016. NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors. *Frontiers in neuroscience* 9 (Jan. 2016), 1–15. `https://doi.org/10.3389/fnins.2015.00516`

[14] Ting-Shuo Chou, Hirak J Kashyap, Jinwei Xing, Stanislav Listopad, Emily L Rounds, Michael Beyeler, Nikil Dutt, and Jeffrey L Krichmar. 2018. CARLsim 4: An Open Source Library for Large Scale, Biologically Detailed Spiking Neural Network Simulation using Heterogeneous Clusters. In *Proceedings of the 2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, Piscataway, NJ, USA, 1–8. `https://doi.org/10.1109/IJCNN.2018.8489326`

[15] Elkin Cruz-Camacho, Siyuan Qian, Ankit Shukla, Neil McGlohon, Shaloo Rakheja, and Christopher Carothers. 2024. Performance Evaluation of Spintronic-Based Spiking Neural Networks using Parallel Discrete-Event Simulation. *ACM Trans. Model. Comput. Simul.* (March 2024).

[16] Andrew P Davison. 2008. PyNN: a common interface for neuronal network simulators. *Frontiers in neuroinformatics* 2 (2008). `https://doi.org/10.3389/neuro.11.011.2008`

[17] E Fehlberg. 1969. Klassische Runge-Kutta-Formeln fünfter und siebenter Ordnung mit Schrittweiten-Kontrolle. *Computing* 4, 2 (June 1969), 93–106.

[18] Andreas K Fidjeland, Etienne B Roesch, Murray P Shanahan, and Wayne Luk. 2009. NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, Piscataway, NJ, USA, 137–144. `https://doi.org/10.1109/ASAP.2009.24`

[19] Richard M Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53. `https://doi.org/10.1145/84537.84545`

[20] Richard M Fujimoto. 1990. Performance of Time Warp Under Synthetic Workloads. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation International, San Diego, CA, USA, 23–28.

[21] Marc-Oliver Gewaltig and Markus Diesmann. 2007. *NEST (NEural Simulation Tool)*. Vol. 2. Scholarpedia, Chapter 4. `https://doi.org/10.4249/scholarpedia.1430`

[22] Marc-Oliver Gewaltig and Markus Diesmann. 2007. NEST (NEural Simulation Tool). *Scholarpedia J.* 2, 4 (2007), 1430.

[23] Samanwoy Ghosh-Dastidar and Hojjat Adeli. 2009. Spiking neural networks. *International journal of neural systems* 19 (2009), 295–308. `https://doi.org/10.1142/S0129065709002002`

[24] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, Z Ghahramani, M Welling, C Cortes, N Lawrence, and K Q Weinberger (Eds.), Vol. 27. Curran Associates, Inc.

[25] Dan F M Goodman. 2009. The Brian simulator. *Frontiers in neuroscience* 3 (2009), 192–197. `https://doi.org/10.3389/neuro.01.026.2009`

[26] Samuel Greengard. 2020. Neuromorphic chips take shape. *Commun. ACM* 63, 8 (July 2020), 9–11. `https://doi.org/10.1145/3403960`

[27] Alexander Hanuschkin, Susanne Kunkel, Moritz Helias, Abigail Morrison, and Markus Diesmann. 2010. A general and efficient method for incorporating precise spike times in globally time-driven simulations. *Frontiers in neuroinformatics* 4 (Oct. 2010), 1–19. `https://doi.org/10.3389/fninf.2010.00113`

[28] Stephan Henker, Johannes Partzsch, and René Schüffny. 2012. Accuracy evaluation of numerical methods used in state-of-the-art simulators for spiking neural

networks. *Journal of computational neuroscience* 32, 2 (April 2012), 309–326. `https://doi.org/10.1007/s10827-011-0353-9`

[29] Suzana Herculano-Houzel. 2012. The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost. *Proceedings of the National Academy of Sciences of the United States of America* 109, Supplement 1 (June 2012), 10661–10668. `https://doi.org/10.1073/pnas.1201895109`

[30] Suzana Herculano-Houzel and Jon H Kaas. 2011. Gorilla and Orangutan Brains Conform to the Primate Cellular Scaling Rules: Implications for Human Evolution. *Brain, behavior and evolution* 77 (2011), 33–44. `https://doi.org/10.1159/000322729`

[31] M L Hines and N T Carnevale. 1997. The NEURON Simulation Environment. *Neural computation* 9 (1997), 1179–1209. `https://doi.org/10.1162/neco.1997.9.6.1179`

[32] Roger V Hoang, Devyani Tanna, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris. 2013. A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. *Frontiers in neuroinformatics* 7 (2013), 10. `https://doi.org/10.3389/fninf.2013.00019`

[33] David R Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 404–425. `https://doi.org/10.1145/3916.3988`

[34] Susanne Kunkel, Maximilian Schmidt, Jochen M Eppler, Hans E Plesser, Gen Masumoto, Jun Igarashi, Shin Ishii, Tomoki Fukai, Abigail Morrison, Markus Diesmann, and Moritz Helias. 2014. Spiking network simulation code for petascale computers. *Frontiers in neuroinformatics* 8 (Oct. 2014), 78. `https://doi.org/10.3389/fninf.2014.00078`

[35] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565.

[36] Mikael Lundqvist, Martin Rehn, Mikael Djurfeldt, and Anders Lansner. 2006. Attractor dynamics in a modular network model of neocortex. *Network* 17, 3 (Sept. 2006), 253–276. `https://doi.org/10.1080/09548980600774619`

[37] L P Maguire, T M McGinnity, B Glackin, A Ghani, A Belatreche, and J Harkin. 2007. Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing* 71, 1 (Dec. 2007), 13–29. `https://doi.org/10.1016/j.neucom.2006.11.029`

[38] Henry Markram. 2006. The blue brain project. *Nature reviews. Neuroscience* 7, 2 (Feb. 2006), 153–160. `https://doi.org/10.1038/nrn1848`

[39] Cyrille Mascart, Gilles Scarella, Patricia Reynaud-Bouret, and Alexandre Muzy. 2021. Scalability of large neural network simulations via activity tracking with time asynchrony and procedural connectivity. *bioRxiv* (June 2021). `https://doi.org/10.1101/2021.06.12.448096`

[40] M Mattia and P Del Giudice. 2000. Efficient event-driven simulation of large networks of spiking neurons and dynamical synapses. *Neural computation* 12, 10 (Oct. 2000), 2305–2329. `https://doi.org/10.1162/089976600300014953`

[41] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D Flickner, William P Risk, Rajit Manohar, and Dharmendra S Modha. 2014. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 6197 (Aug. 2014), 668–673.

[42] Kirill Minkovich, Corey M Thibeault, Michael John O'Brien, Aleksey Nogin, Youngkwan Cho, and Narayan Srinivasa. 2014. HRLSim: a high performance spiking neural network simulator for GPGPU clusters. *IEEE transactions on neural networks and learning systems* 25, 2 (Feb. 2014), 316–331. `https://doi.org/10.1109/TNNLS.2013.2276056`

[43] Abigail Morrison, Sirko Straube, Hans Ekkehard Plesser, and Markus Diesmann. 2007. Exact subthreshold integration with continuous spike times in discrete-time neural network simulations. *Neural computation* 19, 1 (Jan. 2007), 47–79. `https://doi.org/10.1162/neco.2007.19.1.47`

[44] Simon F Müller-Cleve, Vittorio Fra, Lyes Khacef, Alejandro Pequeño-Zurro, Daniel Klepatsch, Evelina Forno, Diego G Ivanovich, Shavika Rastogi, Gianvito Urgese, Friedemann Zenke, and Chiara Bartolozzi. 2022. Braille letter reading: A benchmark for spatio-temporal pattern recognition on neuromorphic hardware. *Front. Neurosci.* 16 (Nov. 2022), 951164.

[45] Quang Anh Pham Nguyen, Philipp Andelfinger, Wentong Cai, and Alois Knoll. 2019. Transitioning Spiking Neural Network Simulators to Heterogeneous Hardware. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS)*. ACM, New York, NY, USA, 115–126. `https://doi.org/10.1145/3316480.3322893`

[46] David M Nicol. 1993. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *J. ACM* 40, 2 (April 1993), 304–333. `https://doi.org/10.1145/151261.151266`

[47] Dmitri E Nikonov and Ian A Young. 2019. Benchmarking Delay and Energy of Neural Inference Circuits. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 5, 2 (Dec. 2019), 75–84.

[48] Daniele M Papetti, Simone Spolaor, Daniela Besozzi, Paolo Cazzaniga, Marco Antoniotti, and Marco S Nobile. 2020. On the automatic calibration of fully analogical spiking neuromorphic chips. In *Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, Piscataway, NJ, USA, 1–8. `https://doi.org/10.1109/IJCNN48605.2020.9206654`

[49] Dejan Pecevski. 2009. PCSIM: A Parallel Simulation Environment for Neural Circuits Fully Integrated with Python. *Frontiers in neuroinformatics* 3 (2009), 15. `https://doi.org/10.3389/neuro.11.011.2009`

[50] Alessandro Pellegrini. 2014. *Techniques for Transparent Parallelization of Discrete Event Simulation Models.* Ph. D. Dissertation. Sapienza, University of Rome.

[51] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2012. The ROme OpTimistic Simulator: Core Internals and Programming Model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS).* ICST, Brussels, Belgium, 96–98. `https://doi.org/10.4108/icst.simutools.2011.245551`

[52] Matthew D Pickett, Gilberto Medeiros-Ribeiro, and R Stanley Williams. 2013. A scalable neuristor built with Mott memristors. *Nature materials* 12 (2013), 114–117. `https://doi.org/10.1038/nmat3510`

[53] Adriano Pimpini, Andrea Piccione, Bruno Ciciani, and Alessandro Pellegrini. 2022. Speculative distributed simulation of very large Spiking Neural Networks. In *Proceedings of the 2022 SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS).* ACM, New York, NY, USA, 93–104. `https://doi.org/10.1145/3518997.3531027`

[54] Adriano Pimpini, Andrea Piccione, and Alessandro Pellegrini. 2022. On the accuracy and performance of spiking neural network simulations. In *Proceedings of the 2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT '22).* IEEE, Piscataway, NJ, USA, 96–103. `https://doi.org/10.1109/ds-rt55542.2022.9932062`

[55] Mark Plagge, Christopher D Carothers, Elsa Gonsiorowski, and Neil Mcglohon. 2018. NeMo: A Massively Parallel Discrete-Event Simulation Model for Neuromorphic Architectures. *ACM Transactions on Modeling and Computer Simulation* 28 (2018), 1–25. `https://doi.org/10.1145/3186317`

[56] Chi-Sang Poon and Kuan Zhou. 2011. Neuromorphic Silicon Neurons and Large-Scale Neural Networks: Challenges and Opportunities. *Frontiers in neuroscience* 5 (2011), 108. `https://doi.org/10.3389/fnins.2011.00108`

[57] Tobias C Potjans and Markus Diesmann. 2014. The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 3 (March 2014), 785–806.

[58] Tobias C Potjans and Markus Diesmann. 2014. The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model. *Cerebral cortex* 24 (2014), 785–806. `https://doi.org/10.1093/cercor/bhs358`

[59] Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pre-training. (2018).

[60] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2021. High-Resolution Image Synthesis with Latent Diffusion Models. (Dec. 2021). arXiv:2112.10752 [cs.CV]

[61] Sanjiv Shah and Mark Bull. 2006. OpenMP. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06)*. ACM Press, New York, NY, USA, 13. `https://doi.org/10.1145/1188455.1188469`

[62] Renan O Shimoura, Nilton L Kamiji, Rodrigo F O Pena, Vinicius L Cordeiro, Cesar C Ceballos, Romaro Cecilia, and Antonio C Roque. 2018. [RE] The Cell-Type Specific Cortical Microcircuit: Relating Structure And Activity In A Full-Scale Spiking Network Model. *Zenodo* 34 (2018), 1537–1557. `https://doi.org/10.5281/ZENODO.1244116`

[63] Athul Sripad, Giovanny Sanchez, Mireya Zapata, Vito Pirrone, Taho Dorta, Salvatore Cambria, Albert Marti, Karthikeyan Krishnamourthy, and Jordi Madrenas. 2018. SNAVA—A real-time multi-FPGA multi-model spiking neural network simulation architecture. *Neural networks: the official journal of the International Neural Network Society* 97 (Jan. 2018), 28–45. `https://doi.org/10.1016/j.neunet.2017.09.011`

[64] Marcel Stimberg, Romain Brette, and Dan F M Goodman. 2019. Brian 2, an intuitive and efficient neural simulator. *eLife* 8, e47314 (Aug. 2019), e47314. `https://doi.org/10.7554/eLife.47314`

[65] Gianluca Susi, Pilar Garcés, Emanuele Paracone, Alessandro Cristini, Mario Salerno, Fernando Maestú, and Ernesto Pereda. 2021. FNS allows efficient event-driven spiking neural network simulations based on a neuron model supporting spike latency. *Scientific reports* 11, 1 (June 2021), 12160. `https://doi.org/10.1038/s41598-021-91513-8`

[66] Ping Tak Peter Tang, Tsung-Han Lin, and Mike Davies. 2017. Sparse Coding by Spiking Neural Networks: Convergence Theory and Computational Results. (May 2017). arXiv:1705.05475 [cs.LG]

[67] David Thomas and Wayne Luk. 2009. FPGA Accelerated Simulation of Biologically Plausible Spiking Neural Networks. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. ieeexplore.ieee.org, 45–52. `https://doi.org/10.1109/FCCM.2009.46`

[68] Jan-Phillip Tiesel and Anthony S Maida. 2009. Using parallel GPU architecture for simulation of planar I/F networks. In *2009 International Joint Conference on Neural Networks*. 3118–3123. `https://doi.org/10.1109/IJCNN.2009.5178688`

[69] Tim P Vogels and Larry F Abbott. 2005. Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons. *The Journal of neuroscience: the official journal of the Society for Neuroscience* 25, 46 (Nov. 2005), 10786–10795. `https://doi.org/10.1523/JNEUROSCI.3508-05.2005`

[70] Runchun M Wang, Chetan S Thakur, and André van Schaik. 2018. An FPGA-Based Massively Parallel Neuromorphic Cortex Simulator. *Frontiers in neuroscience* 12 (April 2018), 213. `https://doi.org/10.3389/fnins.2018.00213`

[71] Esin Yavuz, James Turner, and Thomas Nowotny. 2016. GeNN: a code generation framework for accelerated brain simulations. *Scientific reports* 6 (Jan. 2016), 18854. `https://doi.org/10.1038/srep18854`

[72] Dmitri Yudanov, Muhammad Shaaban, Roy Melton, and Leon Reznik. 2010. GPU-based simulation of spiking neural networks with real-time performance &

high accuracy. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*. 1–8. `https://doi.org/10.1109/IJCNN.2010.5596334`