SAPIENZA
UNIVERSITÀ DI ROMA

High Performance Simulation of Spiking Neural Networks

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Adriano Pimpini
ID number 1645896

Thesis Advisor

Prof. Alessandro Pellegrini

Co-Advisor

Eng. Andrea Piccione

Academic Year 2019/2020

Thesis defended on 22 October 2020
in front of a Board of Examiners composed by:

Prof. Tiziana Catarci (chairman)
Prof. Francesca Cuomo
Prof. Francesco Delli Priscoli
Prof. Stefano Leonardi
Prof. Andrea Marrella
Prof. Alessandro Pellegrini
Prof. Simone Scardapane

High Performance Simulation of Spiking Neural Networks
Master's thesis. Sapienza – University of Rome

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

Author's email: adriano.pimpini@gmail.com

Abstract

Spiking Neural Networks (SNNs) are a class of Artificial Neural Networks that closely mimic biological neural networks. They are particularly interesting for the scientific community because of their potential to advance research in a number of fields, both because of better insights on neural behaviour, benefitting medicine, neuroscience, psychology, and because of the potential in Artificial Intelligence. Their ability to run on a very low energy budget once implemented in hardware makes them even more appealing. However, because of their behaviour that evolves with time, when a hardware implementation is not available, their output cannot simply be computed with a one-shot function—however large—, but rather they need to be simulated.

Simulating Spiking Neural Networks is extremely costly, mainly due to their sheer size. Current simulation methods have trouble scaling up on more powerful systems because of their use of conservative global synchronization methods. In this work, Parallel Discrete Event Simulation (PDES) with Time Warp is proposed as a highly scalable solution to simulate Spiking Neural Networks, thanks to the optimistic approach to synchronization.

The main problem of PDES is the complexity of implementing a model on it, especially of a system that is continuous in time, as time in PDES "jumps" from one event to the next. This greatly increases friction towards adoption of PDES to simulate SNNs. As such, current simulation-based work on SNNs is relegated to worse-scaling approaches. In order to foster the adoption of PDES and further the work on simulation of SNNs on larger scales, in this work a solution is developed and presented that hides the underlying complexity of PDES.

Chapter Organization

In Chapter 1, the research problem addressed in the thesis is introduced and motivations are adduced. In Chapter 2 Artificial and Spiking Neural Networks are introduced, along with other important aspects, in order to frame the research context. In Chapter 3 Parallel Discrete Event simulation, which is the technique on which this work builds the approach, is introduced. In Chapter 4 some well-known

and widespread simulators specialized in Spiking Neural Networks are introduced, to give an idea of what the state of the art is.

In Chapter 5 the problem of simulating large spiking neural networks is introduced and the developed solution is presented in depth, explaining all the actions taken to make PDES transparent to the user.

In Chapter 6 the methods and results of the experimental assessment are presented. In Chapter 7 conclusions are drawn and some directions are suggested for future improvements regarding this work and the research context in general.

Contents

Chapter 1

Introduction

From the beginning of the history of human thought, we have wondered about consciousness: what sets us and other animals apart from a plant? How are we able to reason and have thoughts? Replicating intelligence has always been one of humanity's aspirations, whether it be to study it, to employ it to solve problems, or simply for the sake of playing god.

Thanks to modern research and knowledge, 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 much more. Thus, 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. In recent years, the concept of Artificial Neural Networks (ANNs) has become a hot topic in computer and data science and artificial intelligence, mainly owing to the work of companies such as Google and OpenAI, that succesfully employed ANNs to perform a plethora of tasks with excellent results, from speech recognition, to beating the Go world champion in 2017 ([16], [18], [17]), and beating a team of pro-players of the online multiplayer PC game Dota 2 [11]. Needless to say, the last two are mind-blowing achievements that could have easily earned their place in science fiction novels as recently as 20 years ago.

The Artificial Neural Networks that are the de-facto standard of the industry are however just inspired at a very high level by the way the brain works, and do not really take into account what actually happens inside of it: billions of cells—the

neurons—receive input stimuli in the form of electric pulses, charge themselves up, and when they are "charged enough", they produce an electric impulse themselves, which gets propagated to the other neurons that are connected to them (more on this in Chapter 2). In these ANNs neurons compute a mathematical function in a one-shot fashion every time an input is received, usually with no regard for time whatsoever.

Spiking Neural Networks (SNNs) are a class of ANN that aims to emulate the biological behaviour of the brain. As such, they need to simulate the behaviour of the neuron, synapses, and any other interesting object in real-time. This leads to a higher fidelity execution of the neural network, at the price of higher computational costs. SNNs however show an interesting trait: since the neurons react to and communicate through electrical stimuli, they can be modelled as circuits, and specialized hardware can be implemented that runs SNNs with extremely high performances both in terms of speed and energy consumption. SNNs have been shown to be capable of carrying out tasks that other ANNs do too with comparable accuracy, while boasting an extreme degree of efficiency. Hardware however presents a problem related to building costs: designing and manufacturing a chip is no cheap task, and is not sustainable for prototyping. Neuromorphic chips—such as IBM's TrueNorth [2]—exist, but present limitations related to their design, which constrains experimentation. The applications of SNNs are not limited to AI however, as simulating biologically-accurate networks can be of vital importance for various fields of research, such as Neurology, Neuroscience, and Medicine in general, to name a few. This is why simulation of SNNs is fundamental for the near and the distant future alike. The aim of this work is to make simulation of SNNs viable on High Performance Computing systems.

Current simulation supports for SNNs consist of time-stepped simulators, which carry out the simulation by computing the state of all objects at every small increment of time. These simulators have acceptable performance on single-thread and even multi-thread environments, but lack the ability to scale beyond few computational nodes because of the conservative synchronization methods they employ. Furthermore, updating the state of every object at each timestamp means

updating objects that are doing nothing, too, introducing costs that could be avoided. Parallel Discrete Event Simulation (PDES) [6] with Time Warp (or optimistic PDES) [8] is the simulation method we adopt in this work to make execution on HPC systems possible and worthwhile.

In optimistic PDES, timestamped events are used to mark the passage of time, jumping from one event to the next. When an event is handled by an object, it can generate new events (messages) directed towards other objects in the simulation. The simulation is carried out in parallel on different threads and even nodes, that optimistically schedule events they have locally, assuming no causality violations will happen because of it. If a violation happens, execution is rolled back to a consistent state and resumed. This optimistic synchronization method allows for an extreme degree of parallelism, without having to waste time waiting for synchronization to happen.

The problem with PDES is that current simulators require the user to be familiar with the concept of PDES and sending messages, and managing the object execution. As such, writing a model for PDES is a complex endeavor requiring a high degree of knowledge and familiarity with the approach. This makes developing for PDES very costly, discouraging potential adopters. This holds true for the field of computational neuroscience, too. As such, a simulator that simplifies the adoption of PDES is needed.

To encourage adoption of PDES, in this work a module was developed which, attached to a PDES simulator tailor-made to support Spiking Neural Networks simulation, simplifies the adoption of PDES for simulating Spiking Neural Networks. This is achieved by hiding the complexity behind a series of Application Programming Interfaces that hold the modeller's hand through successful creation and execution of the model. The underlying simulator is multi-threaded and can run on multi-node systems, making execution on HPC systems possible.

Chapter 2

Spiking Neural Networks

The unresting research on Neural Networks as computational systems is the byproduct of the desire to understand and mimic the brain's ability to learn, generalize, and carry out extremely complex tasks. Paired with the incredible efficiency that biological brains have, it is no wonder that the most obvious and sought-after path to achieve such capabilities is that of copying the brain in a number of its aspects, and eventually covering all of them in due detail, when the technology will allow it. Various approaches have been developed, each with its strengths and drawbacks, and each with a different degree of similarity with the original biological structure, with SNNs reaching for a higher degree of fidelity.

The brain is a complex system, composed of a huge number of simple functional units: 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 the synapses to another neuron's dendrite. In Neural Networks which strive for a higher degree of similarity, attention is currently placed on modelling of synapses and their weights, while dendrites are ignored and the axon's presence is abstracted, but its role can still be recognized in topological aspects, such as the neuron's eagerness to

connect to neurons that are geographically closer, and the spike transmission delay, which depends both on the type of synapse and the point of the axon body at which the synapse lies.

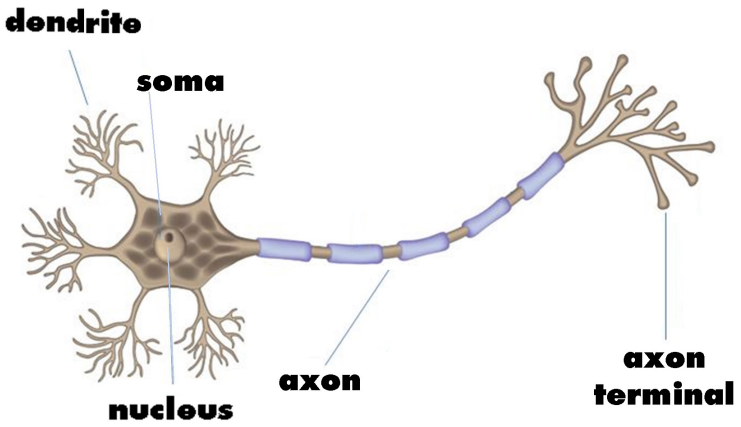


Figure 2.1. Representation of a neuron.
Source: "Neural Networks with R"

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 are sensitive to changes in the membrane 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* these channels are completely closed, however 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* [3] (or signal, or **spike**) and propagates along the body of the cell, specifically along the axon, to ultimately reach the synapses and propagate to the post-synaptic neurons. Note that the depolarization 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.

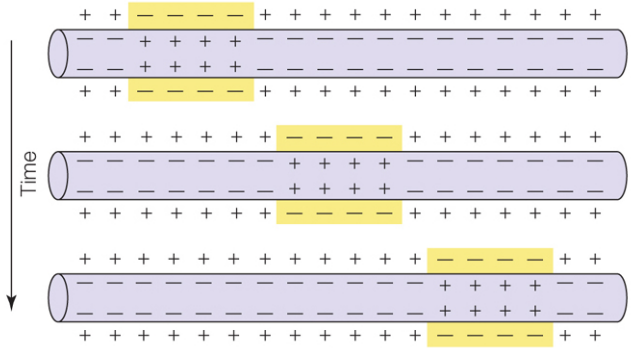


Figure 2.2. A visualization of the action potential propagating through the axon with time.

Source: https://www.macmillanhighered.com/BrainHoney/Resource/6716/digital_first_content/trunk/test/hillis2e/hillis2e_ch34_2.html

Now the cell, that usually has potassium ions inside and sodium ions outside, is back to negative potential, but with potassium ions outside and sodium ions inside the membrane. This situation is reverted by the sodium potassium pump which actively transports sodium back out and potassium back inside of the plasma membrane. 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 "on resetting" process is called *refractory period* of the neuron. These concepts have been introduced as they will be useful in talking about spiking neurons.

In this chapter, Spiking Neural Networks (SNNs) are presented. We begin by introducing the concept of Neural Network (NN), and then go on to take a look at different kinds of NNs and some of their application cases. Lastly we introduce SNNs and go over how they work, their advantages, the challenges that using them presents and how they are currently dealt with, and how they are currently being used to achieve what.

2.1 Neural Networks at a Glance

Artificial Neural Networks (ANNs). ANNs are networks (or circuits) composed of artificial neurons (or nodes). Artificial Intelligence is the field in which ANNs have become so popular thanks to their ability to learn and approximate complex unknown functions. In the simplest kind of ANN for AI—the first generation [10]—, neurons are perceptrons [14]. Perceptrons employ an extremely simple algorithm: every neuron has a vector $\mathbf{w}$ of weights, one for each incoming connection, and a bias b. When the input vector $\mathbf{x}$ is received, the neuron computes the output $f(x)$ based on the value of the following (linear) activation function:

$$f(x) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

The perceptron is a linear binary classifier: a single unit—once trained—can be used to decide whether an input belongs to a class, given that it is linearly separable from the others. A network of perceptrons consisting of three or more layers (one input layer, at least one hidden layer, one output layer) can be built, giving birth to a feed forward ANN commonly referred to as Multilayer Perceptron (MLP). Perceptrons constituting this network usually have a non-linear activation function: networks with this kind of perceptron are seen as the second generation of ANN [10]. The combination of multiple layers and non-linearity in activation allows MLPs to distinguish data that is not linearly separable.

Convolutional Neural Networks (CNNs). CNNs are the class of ANNs that fostered the impressive advancements in Machine Learning and AI we have witnessed in recent years. Because of the way convolution works, they are particularly well suited for images, but the fields of application vary widely. The main differences with MLP lie in the way information is processed, and in the fashion in which neurons are connected. Indeed, MLP suffers from the fact that adjacent layers are usually fully connected, which means a great deal of computation has to be carried out to produce an output, and a fair amount of memory is used to keep track of the weights. CNNs instead are constituted of convolutional blocks. A convolutional block is

made of three layers: convolution, pooling, and activation. The first layer performs *convolution*, which actually is a sliding dot product of the layer's kernel with the input feature maps. Kernels are matrices of weights, which are tuned when training the network. The fact that kernels are layer-specific, as opposed to neuron-specific weights in MLPs, means that there is a much lower number of weights, letting CNNs have a smaller memory footprint. Next is the pooling layer, which is responsible for reducing the spatial size of the representation, so as to reduce the amount of parameters needed and computation done in the network. This is done on every feature map separately, by aggregating different adjacent elements into one. An example is max-pooling: a subset of the feature map is taken and substituted by a single element, the value of which is the max among those of the initial values. Finally, the activation layer, which takes a feature map as its input and outputs a feature map called activation map computed by means of an activation function that is applied element-wise. The repetition of these three processes is the core of CNNs as we know them, and allows CNNs to learn to extract and recognize specific features in the input.

In both the above classes of neural networks, when a neuron receives an input it computes a rather simple function and always yields an output that is instantaneously propagated forward. The computation moves going from the back of the network to the front in a one-shot fashion.

Recurrent Neural Networks (RNNs). RNNs are a class of ANNs in which nodes are organized in successive layers, but differently from the first two classes we have mentioned, the output of every node is directed not only towards the next layer, but also *recurs* by going back towards the neuron that generated it—or some other memory unit—so as to be used in the subsequent timestep. This structure allows them to have a memory of what happened in the past, enabling them to exhibit temporal dynamic behaviour. RNNs are used in prediction tasks, thanks to their ability to analyze time series. Furthermore, they can work on sequences of arbitrary length (contrary to CNNs which require input of predefined length), which makes them prime candidates for text recognition and speech-to-text tasks:

input is fragmented into an ordered number of vectors of the appropriate size, and the vectors are fed to the network, one per timestep (or *frame*). To mention some noteworthy use cases, RNNs are part of what is behind Google's currently unmatched speech-to-text engine, as well as the text-to-speech one.

Similarly to what happens in CNNs, RNNs have a parameter sharing mechanism: while in CNNs weights are shared at layer level and get reused while convoluting over the input feature maps, in RNNs the weights are shared among time steps, thus decreasing the memory footprint, as well as the time needed to train the network.

2.2 Spiking Neural Networks

Spiking Neural Networks (SNNs) are a class of ANNs that mimics **natural** neural networks more closely. This is achieved through the usage of spiking neurons, which communicate by sending signals (spikes) to each other through synapses. Spiking neurons are **stateful**, and the synapses connecting them can be too. Differently from what happens in other classes of ANNs, where neurons produce and propagate an output whenever they receive an input, spiking neurons only fire when a specific condition is met. Specifically, much like biological neurons, they fire 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, which react by increasing or decreasing their membrane potential accordingly, over time. Before reaching other neurons however, the spike passes through synapses, which are weighted and also introduce a **transmission delay**. Indeed, a fundamental aspect that differentiates SNNs from other ANNs is the role that **time** plays: while in the aforementioned classes of ANNs the computation and propagation of the output is instantaneous, spiking neurons need to wait for their membrane charge over time, then when threshold value is reached they fire, and after a transmission delay, only then do the post-synaptic neurons 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. It is worth emphasizing how, while it may look similar in certain aspects, SNNs' time dependence is different from that of RNNs: SNNs evolve through time and keep a memory of the not-so-short past (how long this memory

goes back depends on the physical parameters of the neuron) in their state, while RNNs do so by feeding their output back to themselves—which in SNNs cannot happen—or to other memory units—which are not present in SNNs—to use in the next timestep.

But what, specifically, is a spiking neuron, how is it stateful, and how is biological accuracy achieved? Spiking neurons models are derived from experimental observation of natural 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. We know that neurons' plasma membrane's isolating properties give rise to a capacitance (membrane capacitance C_m), and that 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}. Furthermore we know that in absence of stimuli the membrane potential resets to a resting value V_r; this also holds true after the action potential is generated (which we also refer to as firing, or spiking) 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. Additionally, we know that 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, and that an external current I_{ext} can be supplied (e.g. for experimental observation). This series of observations already gives some clues about what to look for when creating a biological neuron model. Furthermore, the presence of the capacitance alone makes it obvious that a spiking neuron is stateful (the minimum state being just the membrane potential at a given time) and such state evolves with time. This gives a hint on a fundamental aspect that will be discussed later on, around which the entire work presented in this document revolves: to be run on computers, networks of spiking neurons have to be simulated through time. This means that running a SNN on a computer is a costly endeavour.

Why Spiking Neural Networks? If running a SNN is so much more computationally expensive than other ANNs, why should we direct our attention towards them?

The first reason is obvious and possibly already satisfactory on its own: we want to eventually be able to efficiently and precisely simulate a human brain—or parts of it—to be able to study in detail its behaviour in various experiments, or to understand how modifications in the structure or physical aspects of its components would impact it; neurological research would gain a powerful tool to test and validate hypotheses; medicine would be helped in diagnosing and treating brain diseases.

The second reason is that because spiking neurons are modelled as electronic circuits they can easily be implemented in hardware. A series of *neuromorphic chips* have already been created and commercialized (see IBM's TrueNorth neuromorphic processor [2]). 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 machines using Von-Neumann architecture.
- Energy and power efficiency: specialized circuitry is vastly more energy and power efficient than general purpose computational units, whether it be CPUs or GPUs we compare it with.

Sadly, such huge advantages come at a cost, more specifically, hardware manufacturing cost. This is also because SNNs are huge: they require a great number

of neurons, and an even greater number of synapses. Before investing in hardware, it is vital to conduct appropriate research. 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 given, or when implementing a new kind of neuron or synapse that existing neuromorphic chips might not be able to properly replicate. This is where simulation of SNNs comes into play: it is needed to prototype new hardware solutions while staying within a reasonable cost and time frame, and to validate new approaches when hardware solutions are either not available or physically cannot do so. 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 will be able to exploit SNNs to perform some tasks without having a hardware implementation or accelerator.

Simulation. Now that we have established that simulation of SNNs is something that we cannot forgo—not for the short future at least—the time has come to delve deeper into this fascinating world.

Firstly, some words have to be spent on what a simulation is. Simulating a physical system on a computer entails building a mathematical model that describes the dynamic behaviour of said system and approximates it to a satisfactory degree; the evolution through time of such model is then computed (i.e. the model is simulated) with the help of some simulation software, which allows to gain various insights about the real-life system's behaviour under a plethora of different assumptions. Simulation is used when running experiments in real life is unfeasible either because of monetary cost (one might not want to destroy hundreds of airplanes for the sake of seeing what would happen by crashing at different angles), time constraints (even if one had the funds to crash hundreds of airplanes, they would not have the time to produce them), safety reasons (even with the right funds and time, crashing planes remains risky), or straight impossibility (good luck with crashing a plane... on Mars). Furthermore, simulating lets the user extract and keep track of a great amount of

data: depending on the simulation software, the entire state of the simulation can be made available, and every single aspect of the model can easily be measured, leading to easier correctness checks for the model, as well as precise and accessible measurements.

Secondly, the concept of Virtual Time (VT) [8] 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 organizing 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. 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. Virtual time has countless applications and is relevant for a great number of fields, but the point of view that is of interest for this document is that of simulation, and from such point of view it will be dealt with. 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), but, again, not in its evolution: while VT and wall clock time might share the measure unit, the aim when simulating is (generally) to be fast, which means that VT will hopefully go by faster than wall clock time. We will more properly explore the concept and advantages of virtual time in Chapter 3.

Thirdly, the various approaches to simulation need to be touched. The first, most widespread and most intuitive approach to simulation is time-stepped. In this approach, the simulation is carried out by moving the simulation's virtual clock forward one small time increment (step, or tick) at a time: at each tick the entire state of the simulation is re-computed, allowing every component to be updated. This is the most used approach when simulating physics and continuous systems in general (which are usually described and modelled through differential equations), as a plethora of methods exist to approximate differential equations through time

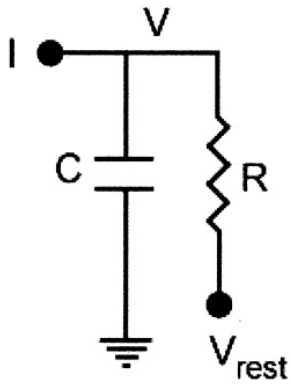
stepping. The main drawback that time stepped simulation has, is that even when nothing happens the computation proceeds at the same exact speed: for every tick, time has to be incremented by one integration step, and all the states in the simulation have to be calculated and updated accordingly. Simulating an object sitting still on a plane for some amount of (virtual) time takes the same amount of wall clock time and computational power that simulating that object bouncing up and down for the same (virtual) time requires. The second approach is Discrete Event Simulation (DES). Events are called discrete after the fact that each event happens at a precise point in time, and its duration is instantaneous with respect to the simulation time's granularity. Phenomena that have a non-impulsive duration can still be modelled by using events that capture their key instants, e.g. when simulating the execution of a system employing the client-server paradigm, a computation that is fast enough to be considered instantaneous can be modelled as a single event, while a longer computation (e.g. one that involves a series of database accesses, rather than a file upload/download) can instead be modelled with two events, one for the start of the action, and another one for its end. In DES the simulation is (actually it optionally can be partitioned, but we assume it always is) partitioned into a series of Logical Processes (LPs), each of which receives events, responds to them by properly handling incoming events and correctly updating its state, and possibly scheduling new ones for the future. For LPs the passage of time happens not with infinitesimal time steps, but rather 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, and in general its adoption to model continuous-time systems is limited, as it can be quite complex.

Simulating SNNs. Having rapidly covered the aforementioned important aspects, we are now ready to take a glance at how spiking neural networks are currently being simulated, and how we approached the problem instead. Both these topics will be discussed more in depth later on in Chapters 3 and 4.

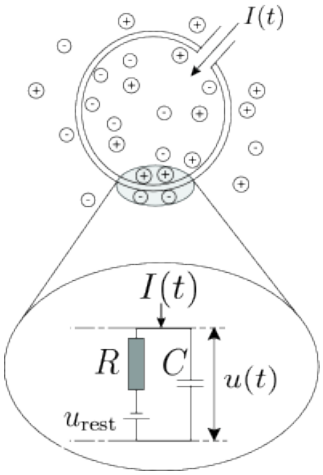
The standard approach to simulating SNNs is time stepped. This might come as no surprise: as mentioned earlier, spiking neural networks are continuous time systems, and time stepped simulation, with its natural affinity to time-stepped approximation of differential functions, is the standard for them. Indeed all of the most prominent SNN simulators (Brian [19], NEST [7], Neuron [5]) are time stepped. Sadly, the ever increasing scale of SNN models has led to manifold increases in simulation times: the run time of time stepping is already severely affected in networks with tens of thousands of neurons and millions of synapses. However, one needs to keep in mind that while spiking neurons are modelled as continuous entities, a great portion of their simulated existence is spent updating their membrane value without actually spiking. Indeed, the membrane potential reaching the threshold potential (and thus spiking) and the reception of spikes from pre-synaptic neurons are the only two *events* worthy of attention: there is no reason to update a neuron knowing that it will not spike, and no reason to wake one such neuron up (and compute its state) unless it receives a new incoming spike. We see how spikes are the embodiment of a discrete event in time: a spike either happens or does not and, if a spike takes place, there is only a distinct moment in time at which a spike is fired, and one time at which it gets delivered. With such considerations, we see how DES starts becoming a valid candidate to simulate this kind of networks. SNNs are indeed one of those cases for which, while time-continuous, the trajectory of the state evolution is known after each event, and can thus be modelled by only taking into account the key events that are spike reception and generation. Namely, when an event is received at time t_{now}, the state needs to be updated from when it was last updated t_{lu} (i.e. when the neuron last received a spike or gave rise to one) to t_{now} by taking the correct point of the trajectory $T_{lu}(t_{now})$; then the effects of the event have to be applied to the neuron, computing a new trajectory $T_n(t)$ that the neuron state will follow starting from t_{now} on until it either spikes or receives a new input. This obviously gives the best results when a closed form solution for the state trajectory is known, but can also be achieved through algorithms for approximate resolution of the differential equations constituting the model.

2.3 The Leaky Integrate and Fire Spiking Neuron

One of the simplest spiking neuron models is the Leaky Integrate and Fire (LIF) neuron. First developed by L. Lapicque in 1907 [1], it models the neuron as a leaky integrator. The LIF neuron's extreme simplicity allows for an extremely lightweight representation of a neuron, from the standpoints of both computational and memory costs. This has earned it the spot as one of the most used neurons for SNNs for deep learning and artificial intelligence.



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



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

Figure 2.3

The LIF model (in Figure 2.3) comprises all the fundamental aspects we have mentioned at the start of Section 2.2: (looking at Figure 2.3a) A 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]). What is missing from the model however, is the

refractory period.

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

$$I(t) = I_R + I_C \tag{2.2}$$

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

$$I_R = \frac{V_R}{R} \quad \text{where} \quad V_R = V(t) - V_{rest} \tag{2.3}$$

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

$$I_C = C\frac{\mathrm{d}V}{\mathrm{d}t} \quad \text{because} \quad C := \frac{q}{V} \quad \text{and} \quad I_C := \frac{\mathrm{d}q}{\mathrm{d}t} \tag{2.4}$$

From Equations 2.3 and 2.4 we get:

$$I(t) = \frac{V(t) - V_{rest}}{R} + C\frac{\mathrm{d}V}{\mathrm{d}t} \tag{2.5}$$

By multiplying Equation 2.5 by R we obtain the standard form:

$$\tau_m \frac{\mathrm{d}V}{\mathrm{d}t} = -(V(t) - V_{rest}) + RI(t) \tag{2.6}$$

Where $\tau_m = RC$ is the membrane time constant of the neuron. Equation 2.6 is the differential equation that models 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 theorized 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.

As mentioned, the Leaky Integrate and Fire neuron model has its evident limitations. It neglects a series of aspects of neuronal dynamics; in particular the fact that there is no mention in the mathematical model of the threshold voltage, which could be expressed with the following condition:

$$\text{if} \quad V(t) = V_{th} \quad \text{then} \quad \lim_{\epsilon \to 0^+} V(t+\epsilon) = V_{rest} \tag{2.7}$$

Still, the refractory period would be ignored. Refractoriness of a neuron that recently spiked might be expressed as (with τ_{ref} the refractory period duration):

$$V(t) = V_{th} \wedge \hat{t} \in (t, t+\tau_{ref}] \implies V(\hat{t}) = V_{rest} \tag{2.8}$$

Note that Equation 2.8 implies Equation 2.7 for $\tau_{ref} > 0$. As such we could use Equations 2.8 to extend Equation 2.6 to obtain a model with refractory behaviour.

While a model as simplistic as leaky integrate and fire is obviously an approximation of a true neuron's behaviour, Lapicque's work certainly 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.

Chapter 3

Parallel Discrete Event Simulation

In this chapter, Parallel Discrete Event Simulation (PDES) is discussed, starting from DES, to then go on to explore the challenges that parallelizing DES poses and how they can be properly faced. What follows stems from knowledge accrued through practice and through the study of various scientific papers on the topic, among which sections of [12] have been used as starting point and ground truth.

3.1 Discrete Event Simulation

In Section 2.2 Discrete Event Simulation (DES) was first introduced in this document, along with a few key concepts:

- Events are called Discrete because of their impulsive nature.
- The simulation can be partitioned into a series of sub-processes called Logical Processes (LPs) which receive events, process them, and possibly raise new events. We also stated that (for the sake of continuity) we will assume that it always is partitioned into LPs.
- The concept of Virtual Time (VT), that is the logical time associated with the simulation, and Wall Clock Time (WCT), the time of real life.

- In DES (and its LPs) the passage of time is not continuous, but rather time skips from one event's time to the next event's. What happens between events is either irrelevant or known.

The time has now come to talk about DES with a more well-formed, systemic approach.

3.1.1 Systemic Approach to DES

When a systemic approach to DES is adopted, the model is seen 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 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 (it should be emphasized one last time, that events take place at a precise time instant).

It appears clear how DES borrows its approach from event-driven programming, in which the execution of the program is determined by events and the actions taken in response to them. Indeed "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." ([12]).

While handling an event e, a series of new events can be generated. This brings to our attention the need to talk about the notion of causality between events: if an event e' is generated while processing event e, we say that e' *causally depends* from e. This means that e' only exists as a consequence of the execution of event e, as well as all the events that will causally depend from e', since causal dependence is very obviously a transitive property. Furthermore, as anticipated when talking about

DES for the first time, due to the non-decreasing nature of (virtual) time it has to hold that for all events e' causally dependent from e we have $T_{e'} \geq T_e$. This seems rather obvious as events happening in the present or future shall not influence the past. The notion of causal dependence may sound similar to the happened-before relationship in distributed systems [9]; indeed the languages revolving around these two worlds are very alike as they face challenges that are actually fairly similar, as such it is no surprise that defining, understanding, and enforcing causal dependence is extremely important to guarantee correctness of the execution, especially in PDES.

In the systemic vision of DES, the simulation can be split into its 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 would need to decide which of the two events has to be passed to the handlers first; this is done through a tie-breaking policy. These policies have a very delicate job, as malformed tie-breaking could lead to biased or erroneous simulation.

3.1.2 Components of DES

We have mentioned that the kernel and the model have a series of duties to fulfill. This is achieved through their cooperation using a series of components that will now be explored.

Simulation state. As mentioned above and often repeated in this document, 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—no wonder—the core concept of the whole DES approach. Each simulation model takes care of defining a-priori the set E of event types that might take place at run time. This definition process also entails defining an appropriate programmatic event handler for every event $e \in E$. The only event that is not defined by the model is the initial event, typically called INIT. It is scheduled by the kernel before starting the simulation, has timestamp 0 and serves the purpose of letting the model initialize all the needed variables and data structures representing the state to their initial values and schedule the first batch of events. While automatically generated, a handler must be defined in the model for INIT, as the initialization is model-dependent. A closing event, say, DEINIT, could also be raised when finishing the simulation to notify the LPs to perform any action they might want to upon closing; however this could be seen as unorthodox.

Clock. While keeping track of Wall Clock Time is responsibility of the Operating System, the simulation kernel has to keep track of the Virtual Time of the simulation. As we know, in DES VT does not move continuously, but rather leaps from one event's timestamp to the next event's one. 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, but this will be discussed later.

Event Queue. The kernel is tasked with keeping track of events and delivering them at appropriate times. Since the number of future events is generally greater than 1, this is achieved through the use of the Event Queue (EQ). The event queue is a priority queue that holds events that are scheduled for the future: when the model wants to schedule a new event, it generates it and calls the appropriate kernel API; the kernel receives the event and puts it into the queue. When the handling of an event concludes (i.e. the handler function returns control to the kernel), the

kernel must extract the next upcoming event from the EQ, this means the event queue must be ordered by event timestamp, so that extracting the next event to be processed is a fast operation. The actual implementation of the priority queue is irrelevant, as long as its performance is good for the application, however the event queue is definitely write-heavy, and this has to be taken into account when choosing the implementation. The sorting aspect is important as the framework needs to always pick the event with the lowest timestamp that sits in the EQ to be raised next. Given with 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. Luckily, given that DES is serial the state is entirely visible at any moment, and that the simulation is stopped when in kernel mode, raising events in order is not a challenge.

Ending Condition. Simulation models often describe phenomena that do not stop on their own, but rather continue on evolving. Programmatically, this means that they keep generating events for as long as their event handlers keep getting called, never letting the event queue empty out. Also, there are phenomena for which the ending time is not known a-priori. This means that we need to be able to stop the simulation once a given ending condition is met.

This can be done either by specifying a maximum time for 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. The first approach can be carried out entirely by the simulation kernel, 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. In both cases this happens through specific APIs.

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 (i.e. to parallel and/or distributed architectures).

3.1.3 DES Kernel logic

We introduced DES, along with all the components that allow the kernel and model to run the simulation. We have made references to how a kernel carries out its duties. In this section the basic logic behind a discrete event simulation kernel is presented.

As explained in Section 3.1.2, the kernel's job is to manage events and deliver them at the correct time instant. This process is kick started by the scheduling of the *INIT* event during the kernel initialization. An extremely stripped down version of the kernel's implementation is shown in Algorithm 1. The two procedures, INIT and SIMULATION-LOOP take place.

Each consists of a few basic operations. We start from the INIT procedure:

1. The ending flag is set to false. This is to take into account the ending condition.
2. The simulation State is initialized, by allocating its memory, gathering the seeds for random number generation, and whatever further action the specific implementation of the kernel might need.
3. The simulation Clock is initialized, by setting its value to the initial time (generally, $VT_0 = 0$).
4. The INIT event is either scheduled, so that the simulation loop can then pop it from the event queue and feeds it to the handler, or the handler for the INIT event is directly called.

And move on to analysing the main loop SIMULATION-LOOP, which is entered immediately after the INIT procedure ends:

1. The while cycle is entered and it continues until the ending condition is met.

2. The next upcoming event e_n (i.e. the one with the minimum timestamp among those in the EQ) is extracted from the Event Queue.
3. The simulation Clock is set to T_{e_n} to actualize the VT advancement.
4. The correct handler for event e_n is called with the event as parameter. If LPs are used, the correct handler for the target LP is invoked.
5. The model is now in charge and:
   - The event is executed.
   - New events are possibly scheduled.
   - The simulation state (or the LP state) is updated.
6. Optional step: Statistics are gathered and updated
7. The ending condition is checked and the flag updated.

In the optional step, besides gathering statistics and logging kernel-side, a model-side function can be called to let the model gather its own statistics and log them to storage.

3.2 Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) [6] refers to the execution of a single Discrete Event Simulation on a parallel/distributed system. This requires for the DES program to be transported into a PDES program. Since—as we have mentioned in Section 3.1.2—DES can be seen as two components interacting (the simulation model and kernel), work can be conducted on each of the two components separately to enact this transformation. While models can easily be ported to the parallel paradigm provided they respect a set of restrictions, the simulation kernel has to undergo a series of important modifications. This however means that once the work on the simulation kernel is done, DES models can be adapted to the PDES paradigm with very limited to no efforts.

While in DES the concept of Logical Process described in the previous sections is optional, in PDES it becomes vital. The name Logical Process is the result of

Algorithm 1: Skeleton of DES execution. Source: [12]

```
procedure: INIT
    End ← false
    initialize State, Clock
    schedule INIT
end procedure

procedure: SIMULATION-LOOP
    while End == false do:
        Clock ← next event's time
        process next event
        Update statistics
    end while
end procedure
```

a parallel between LPs in simulation, and processes in operating systems. Indeed LPs operate in an independent manner one from the other, and without any shared memory: their only way to communicate is through messages (i.e. events). A simulation is then composed by N LPs, each of which has its own unique identifier number assigned in $[0, N-1]$. They will be called $LP_0, LP_1, ..., LP_{N-1}$. While in DES there was no need for it, in PDES each LP has its own private clock, referred to as Local Virtual Time (LVT), as opposed to Global Virtual Time (GVT), which is the time of the committed simulation. The adoption of LVT means that two different LPs (e.g. LP_i and LP_j) can have different LVT values, $LVT_i \neq LVT_j$.

From this decoupling between the simulation shards that LPs are, comes the first, most important restriction: no shared variables may be used in the model. 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 clearly is not acceptable. Thus, the model programmer is required to split the

simulation state S into per-LP subportions S_i such that:

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

Equation 3.1 also implies that the simulation state is the union of LP local states, meaning that no part of the state is left non-assigned to any LP: no global variables can be used. The restriction on global variables can pose no problem for some applications, but can also be troubling for others, forcing data organization with sub-optimal performance, or requiring a great amount of messages to be exchanged.

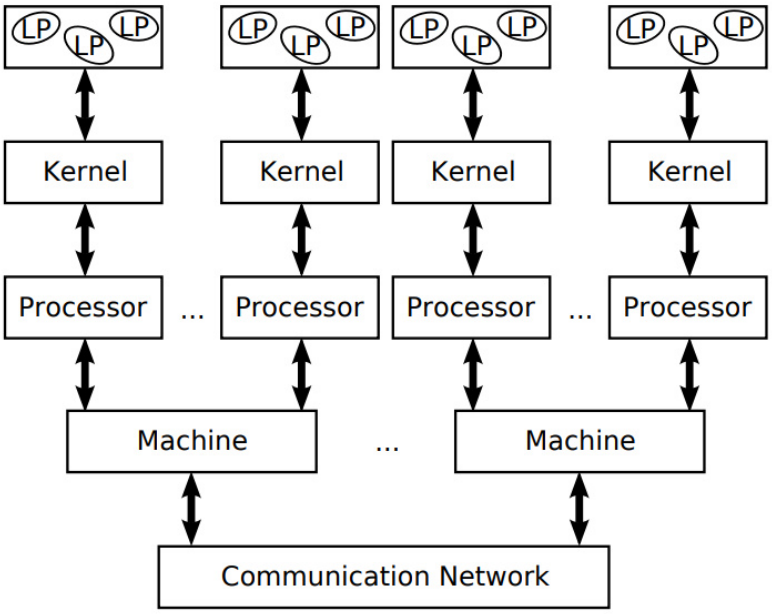


Figure 3.1. Classical Architecture of a Parallel Discrete Event Simulator.
Source: [12]

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 is 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 that is put on the *distributed memory* that is in place to communicate with instances that are actually remote. 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 local LPs even more.

The removal of memory separation brings along the concept of *LP binding*, that is, which worker thread is assigned which LPs to manage. This of great importance because the binding is relevant for memory locality matters, but also allows to easily perform *load sharing* [20], [21] by moving LPs among local threads, requiring no memory costs other than the cache related ones. An action more lightweight than load balancing, which involves *LP migration*, which is an extremely costly operation that requires careful evaluation.

3.2.1 The Synchronization 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 1, 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 *synchronization problem*.

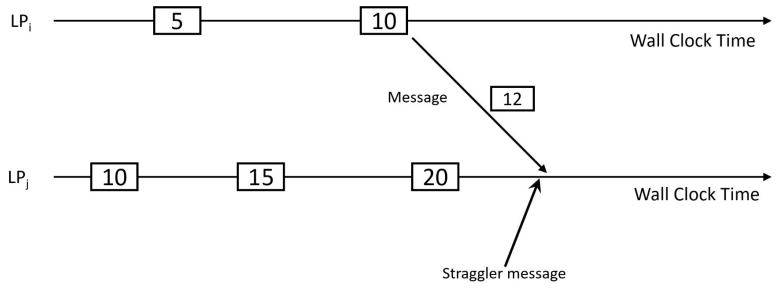


Figure 3.2.
Source: [12]

As it is inescapable, the synchronization 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 errors altogether by executing an event only once it is considered to be safe. The optimistic category executes events "optimistically hoping" that no causality violations will happen. If any such violation happens, then correcting actions are taken. Both 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 synchronization.

3.2.2 Optimistic Synchronization

In optimistic synchronization, *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 synchronization problem was first presented in [8], 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 synchronization 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, we need 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, especially 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.

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

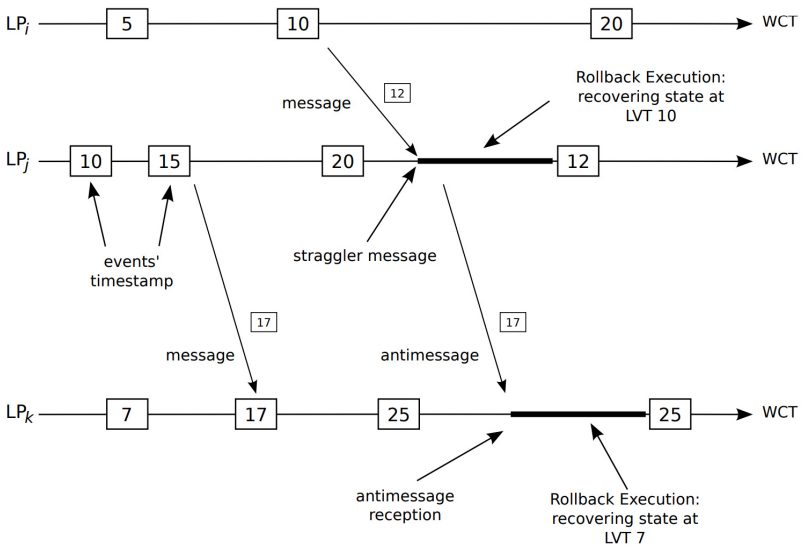


Figure 3.3. An execution in which a rollback takes place.
Source: [12]

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 undoing of messages is achieved through antimessages. Antimessages are copies of already sent messages with a negative sign. When rolling back, the LP checks if any message was sent during speculative 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 the message the antimessage annihilates is still in the Event Queue. In this case the effect of the antimessage is the removal of the message from the queue.
- 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 straggler message can cause a series of rollbacks because of wrong assumptions. 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 [8], 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, according to the following definition:

Definition 3.2.1. Global Virtual Time. GVT(t) is defined as the minimum timestamp of any unprocessed message or anti-message flowing in the system at Wall Clock Time t.

Definition 3.2.1 tells us how to compute GVT(t), that is by inspecting the timestamps of all the unprocessed messages currently in the system. Due to the distributed nature of PDES, this also means that we need to keep into account 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. Now that we have the concept of GVT, *State Save & Restore* can be further explored. As mentioned, the idea is that of taking a snapshot of an LP's simulation state and keep it stashed so that if a rollback needs to be issued, the state can be restored using a snapshot that is still correct. Naturally, a series of snapshots is taken for every LP.

Supposing that a snapshot is taken after processing each event, when a straggler message with timestamp $T_{straggler}$ arrives 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 $T_{snapshot} < T_{straggler}$.

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 straggler is received with timestamp $T_{straggler}$ and a snapshot with $T_s < T_{straggler}$ is restored following the rollback, if there happen to be an event e with associated timestamp $T_s < T_e < T_{straggler}$, its execution will have to be carried out again before continuing. As such, the GVT period has to be chosen (or computed at run time) to be one that minimizes the amount of time spent creating snapshots and re-executing correct events that were rolled back because of snapshot sparsity.

For the memory usage, GVT comes to the rescue: if a state has been committed it is obviously correct and will never not be. As such, all the snapshots with timestamp $T < GVT(t)$ can be discarded as not needed, exception made for the most recent one which always has to be kept. This process is called *fossil collection*.

3.2.3 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.

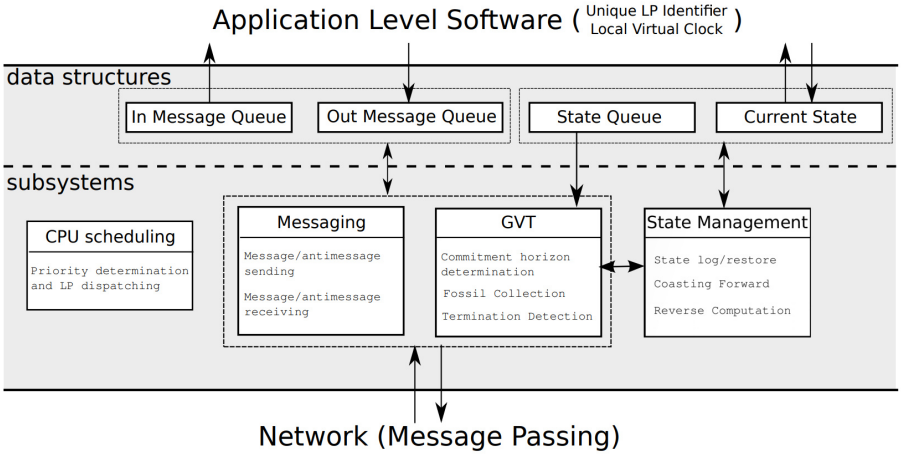


Figure 3.4. A Reference Architecture for Optimistic Simulation Systems.
Source: [12]

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*, that is reprocessing the intermediate events between the restored snapshot and the timestamp of the straggler message, but without sending out messages (as the antimessages have not been sent for such events, as they were correct).
- Performing fossil-collection, i.e. discarding values with timestamps older than current GVT (except from the most recent one, as always).

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 seed inside the snapshots.

Chapter 4

Related Work

In this chapter, the state of the art in simulation of spiking neural networks is presented.

The world of SNN simulators is wide and variegated, in this work the simulators that appear to be most widely used are presented. These simulators are:

- Brian [19], an open-source SNN simulator written in Python. Designed with ease of use in mind.
- Neuron [5], which is also open-source. Boasting more than 1900 articles reporting that work was done with it, it is oriented to precise, in-depth simulation of neuron models, but also supports larger-scale simulations. Cooperates with the Human Brain Project.
- NEST [7], open-source as well. With over 290 research papers citing it as the simulator on which the code was run.

It is important to note that all of the simulators mentioned above employ the **time-stepped** simulation paradigm.

4.1 Brian

Brian [19] is an open-source time-stepped simulator written in Python. Its main focus is ease of use, but sports a series of interesting facilities that allow for unlimited flexibility and high performance.

The simulation structure is simple and follows few key steps:

1. The runtime is intialized automatically upon importing Brian.
2. Neurons are declared and initialized through the NeuronGroup class.
3. Synaptic connection between two NeuronGroups is declared with a SynapseGroup object.
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 (solves model equations) is written and compiled on-the-fly when initializing the simulation. Based on the model, high-level code in string format is generated, which are then transposed into an intermediate representation that is optimized and then compiled into fast, low-level C++ code. While this speeds Brian up, the current lack of multi-threading can be seen as an impairment. However, the possibility to use OpenMP for multithreading when running in C++ standalone mode is being worked on.

The extremely high-level interface that Brian exposes is no doubt a great feature that allows modellers to rapidly develop and test models without extensive knowledge in computer science. Coupled with runtime code generation, it is able to deliver a high standard of performance and can easily be called ground-breaking as far as user experience goes.

Another interesting facility that Brian provides is that of creating groups of neurons the only purpose of which is generating input, such as a PoissonGroup, which generates a number of poisson neurons that are entirely managed by the simulator, while being addressable by the user as normal neurons in a NeuronGroup. The abstraction level is important, meaning that Brian is intended to simulate the high-level behaviour of networks and their neurons in order to measure parameters such as firing rates, rather than ion concentrations (even though one such thing could be modelled through more differential equations). This makes it more useful for observing ensemble behaviour of neural networks, rather than fine-grain neuron details.

4.2 Neuron

Neuron [5] is an open-source time-stepped simulator written for the most part in C and C++. Its main focus is on biologically accurate simulation. Neuron is built with fine-grain details in mind: it allows the modeller to model, simulate and observe most of the imaginable aspects of neurons and synapses, starting from electrical, to chemical, to topological. The simulator can describe ion concentrations and the functioning of ion gates, the dynamics of the ion diffusion, and more. The geometry library allows for a 3D representation of neurons to be built, allowing to model various parts of the cell (that can even be split further into subsections) soma, dendrites, axons, along with their 3D position and orientation, both in the neuron, and in space. This all can be done either programmatically, or through the CellBuilder GUI (Figure 4.1), which even lets the user import a 3D reconstruction of the cell one wants to model.

Neuron boasts a huge number of already implemented, ready to use neuron, synapse, and network models. These—or any custom entity whether it be a neuron,

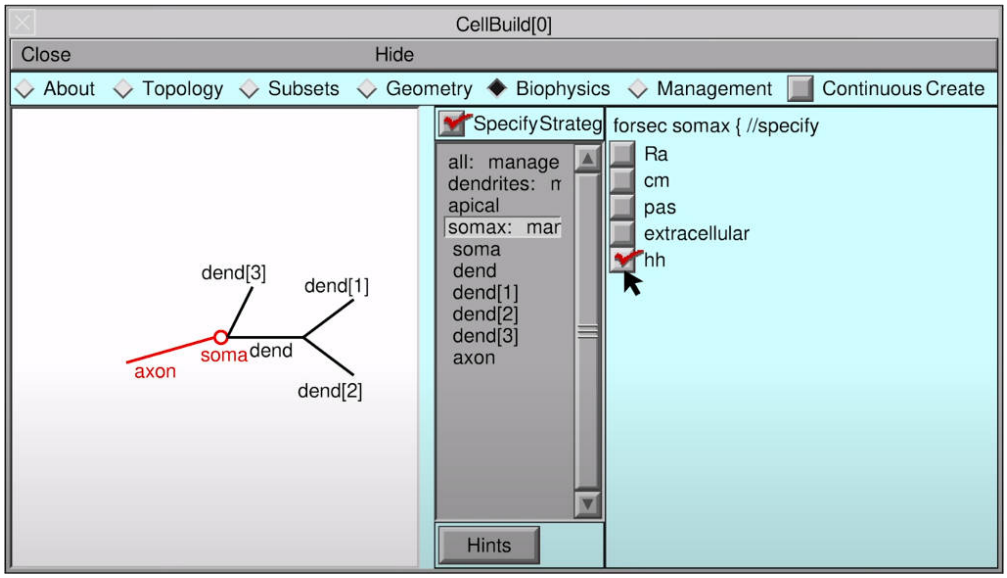


Figure 4.1. Neuron's CellBuilder GUI.

synapse, or entire network—can be loaded either programmatically or with their GUI (Figure 4.2), which lets the user set the parameters for the experiment and run them. The GUI approach is obviously more suitable for "playing around" to get to know models and to observe their behaviour, or to verify that the behaviour of a newly implemented model is correct.

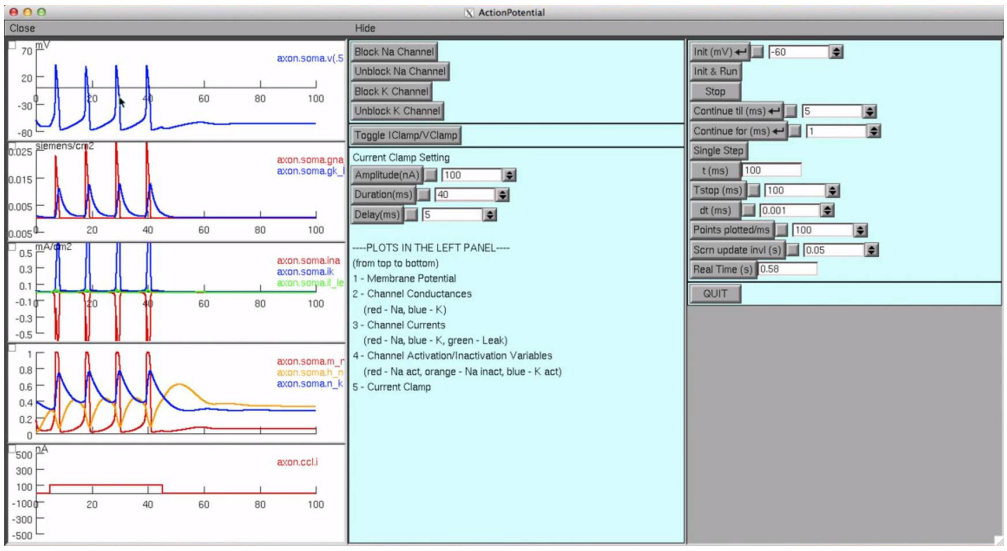


Figure 4.2. Neuron's GUI when running an experiment.

All of the aforementioned characteristics make Neuron a simulator that is most

well-suited for smaller simulations with high biological precision, as opposed to the "fast and dirty" approach that is taken by simulators that are used when examining the ensemble behaviour of SNNs, for which a higher-abstraction is pursued with respect to single-neuron dynamics, in favor of network simulation size.

Notwithstanding this natural predisposition towards smaller simulations that Neuron has, it can simulate fairly large networks, and most importantly can run simulations on clusters, using MPI to abstract the network communication. However when running on multicores, Neuron does not exploit the worker thread paradigm, but the various kernel instances are run as separate processes, and MPI is used to connect them, leading to a worse performance with respect to what would be achieved using worker threads and using MPI only to reach physically remote nodes.

When the simulation is run in a parallel and/or distributed environment, generated spikes need to be disseminated in the system, as such a conservative approach is taken, in the form of a maximum simulation time lookahead, which is equal to the minimum synaptic delay among all the incoming synapses coming from another computing node. As such, based on synaptic delays, a series of synchronization points is created at which all the scheduled events are exchanged between computing nodes. This is a very costly but necessary operation, however it is conducted sub-optimally because it uses the communication network in bursts rather than maintaining a steady pace, which causes moments of congestion when exchanging messages, alternated to periods of silence. A better approach would be, while maintaining the constraint of maximum simulation lookahead, exchanging messages whenever they are generated in order to keep a steady but flowing network usage.

Neuron has a Python interface for programmatically instantiating networks and running simulations. Membrane (and other) dynamics can instead be defined using a Domain Specific Language.

4.3 NEST

NEST [7], the last of the three simulators presented in this chapter, is also open-source and time-stepped, and written in mostly C++. As the other two simulators it has Python interfaces to increase ease of use, but comes prepacked with "over 50

neuron models many of which have been published" and "over 10 synapse models", and users can implement new custom neuron and synapse models.

NEST can run parallel simulations through OpenMP, which means that it adopts the worker thread paradigm when performing parallel computing, enjoying the benefits of a more lightweight simulation kernel. Distributed simulations are also supported and MPI is used to take care of message passing between multiple computational nodes. Although OpenMP is used, message passing is always handled by the global spike exchange mechanism, which here we take as to mean that messages pass through MPI even when directed to another worker thread of the same machine. The inter-process communication and node distribution is handled transparently inside of NEST.

Neurons are instantiated only on the node on which they belong (which also happens in Neuron), while devices are duplicated on every kernel instance. Synapses are handled at the receiving node's end for matters of synapse plasticity.

Chapter 5

Simulating Large Spiking Neural Networks

PDES is a complex paradigm, which makes building simulators hard and costly. (P)DES simulators usually do not offer a high level interface that abstracts from the underlying presence of the Discrete Event Simulation. This means that model writers need to be at least familiar with the concept of event, and how the time passes inside a DES simulation. Then all of this has to be handled, i.e. neuron spikes have to be mapped to events, and events are received and converted into spikes. Then there are control events: when in PDES, a neuron cannot move forward in the future. This means that it is "chained" to the virtual time of the event. This means that it cannot explore the future and decide that it will spike at time T_{spike}. What the neuron can do is evaluate, given the current state (membrane voltage, input current, etc.), whether it will spike in the future, and if so, when. Once this is done, then the neuron could schedule an event to wake itself up and check if its state variables are such that it indeed meets the spiking conditions. There is no need to say that getting to the point in which the need for this cumbersome process becomes evident already requires having spent enough time to at least understand what happens during a DES simulation and how to simulate a continuous time system in a discrete time environment. This alone would be enough to discourage any individual without good knowledge of PDES, or at least computer science, from trying to implement their model for a PDES simulator. What this is saying is that

development cost is the main problem with current DES simulators, when compared to their time-stepped competitors: it already takes a long time to become an expert in computational neuroscience, having to become also proficient in computer science is a cost not many (if anyone) are willing to pay. Indeed as if having to deal with all the discrete event world was not enough, the problem is that the model code needs to be error-free but, most importantly, it has to be efficient, as it can easily become the bottleneck of the execution. It is important to remember that if no facility is exposed for SNN support, the modeller also needs to come up with ways to handle synapses: their representation, their behaviour when simulating, whether they are to be modelled as LPs themselves (they probably should not if we are looking to simulate an SNN for AI and not for extreme biological accuracy—and even in that case it could be debatable whether a single synapse should have the dignity of an entire LP), and scheduling spike events by looping on all of them, as well as possibly waking itself up after spiking to see if another spike could belong in the neuron's future.

Saying that this is as complex as it sounds would mean understating the real complexity of the whole endeavor. A simulation framework built to support spiking neural network simulation on parallel discrete event simulator is needed. One such framework would need to hide all the idiosyncrasies related to PDES by providing an API, no matter how extended, that guides the modeller in the implementation process, making it easier for someone that is not a computer scientist to build a SNN model for discrete event simulation.

The modeller should not need to worry about where the synapses are kept or how they are organized, nor should they care about the fact that spikes are in fact events that need to be sent one-by-one and, 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. In this work, a series of important steps have been undertaken to render the modelling more easily achievable, requiring that the modeller only:

- Implement a series of APIs that the simulation framework calls in each specific circumstance.

- Implement a function that can compute the next presumed spike time and communicate it to the framework, which then takes care of verifying that the spiking conditions are still met once the time comes.

To achieve this, a module has been developed that is hooked into the simulator.

5.1 The module

To help modellers more easily develop Spiking Neural Network models to simulate through PDES, in this work a module has been developed, which is attached to the simulator and hides its complexity from the user. The modeller is supposed to only use the module's interfaces to execute the simulation, so as to get guided by the module itself, and remove the need to manage events and the like.

5.1.1 Interfaces

The module encapsulates the complexity of the Discrete Event Simulation with the interfaces that follow.

Spiking. When a neuron wants to spike, there are two kinds of interface exposed by the framework:

- SendSpike
- MaybeSpike and MaybeSpikeAndWake

SendSpike allows to simply schedule a spike for a given time in the future. It is translated by the framework into a series of messages, one per outgoing synapse, directed to the postsynaptic neurons, where each message carries the weight of the synapse.

MaybeSpike and *MaybeSpikeAndWake* are functions that handle the hypothetical nature of spiking in the future, as such they allow the user to schedule a spike that will take place if and only if the calling neuron receives no other spikes in the meantime. This is achieved by scheduling a MAYBESPIKE or MAYBESPIKEWAKE event to wake the module at the time of the hypothetical spike. For each LP, the

framework keeps track of a progressive identifier. When MaybeSpike/AndWake is called, upon scheduling the MAYBESPIKE/WAKE event, its payload is set to the current identifier. Whenever an event other than MAYBESPIKE/WAKE is delivered to the neuron, the identifier is incremented. When a MAYBESPIKE/WAKE event is received, the framework checks its payload: if it matches with the current identifier, it means that no other event has taken place in the meantime and the event has to be executed, otherwise it is discarded (ignored) (note that wrapping around of the identifier has not been deemed an issue as 64 bit integers are used). When a valid MAYBESPIKE/WAKE event (with payload matching the current identifier) is executed, the spike is materialised in the same fashion as a SendSpike, with the very same function being called. If the event is a MAYBESPIKEWAKE, then the spiking neuron is also woken up by calling the NeuronWake interface. Waking the neuron after spiking is useful for self-spiking neurons, or to check whether a neuron will have enough input current to spike again even without being solicited from other neurons, and as such possibly invoke MaybeSpike/AndWake again.

Initialization. To initialize the network, a series of interfaces are exposed:

- NeuronInit
- SNNInitTopology
- NewSynapse
- NewSpikeTrain
- NewProbe

NeuronInit needs to be implemented by the user. When invoked, it is given the identifier of the neuron to be initialized. The user is expected to create the neuron state in it through malloc, and return a pointer to said state.

SNNInitTopology is also implemented by the user. In this function, raised after all neurons have been initialized, the topology is initialized. The user may use NewSynapse, NewSpikeTrain, and NewProbe to build the network.

NewSynapse is used to create a new synapse between two neurons. It works as a wrapper to malloc, allocating the memory for a synapse's state and returning a pointer to such memory. The allocation is only successful if the synapse lies on the current node, and more specifically if the source neuron is managed by the current thread, as synapses are managed by their pre-synaptic neuron. The synapse can be either dynamic or static, meaning that its state can either change (and thus needs to be tracked for performing rollbacks) or not, allowing to further decrease the load put on the state management subsystem, and thus memory and computational costs associated with it. Considering that each neuron has a great number of outgoing synapses associated to it (in the model we ran, with circa 80'000 neurons, each neuron had more than 3000 synapses), properly exploiting the possibility to declare synapses as static allows for a huge boost in performance. This said, all the user needs to do is to invoke NewSynapse and check that the returned pointer is not NULL, if that is the case, the synapse can be initialized normally.

NewSpikeTrain is used to create spike trains, which are series of spikes directed to specific neurons at specific times. Spike trains are used as inputs to Spiking Neural Networks. The implementation is straightforward: the module schedules one event per spike per target neuron. These events are not managed in any LP's outbound queue as they are not generated by any of the neurons, and need not be rolled back through antimessage as they are not speculative. This is achieved by creating and inserting the events in the thread's input queue through a custom function that does not push the messages in the outbound queue of any LP. Each thread only schedules messages directed to LPs it owns when it executes a NewSpikeTrain. The events are managed correctly by the messaging subsystem, meaning they can be rolled back and re-scheduled, and are discarded once $GVT > Timestamp$.

NewProbe allows to probe a neuron, meaning that it is monitored and each time said neuron fires, the user-defined ProbeRead handler is called, to extract any interesting information. Furthermore, probed neurons are the only ones that are interrogated when evaluating the ending condition through NeuronCanEnd. Tracking probed neurons happens through a simple flag.

Advanced memory management. For users that may want to create memory areas that are read only (e.g. the physical parameters of the neuron), an interface to allocate static memory is provided. This memory is called static because it will not get checkpointed nor rolled back by the framework. This is obviously only safe when the allocation and initialization with data are performed during the network initialization—before running the actual simulation—, as it could lead to inconsistencies otherwise. Using static memory for appropriate memory areas relieves from part of the work from the framework, as no attention needs to be payed to them by the rollback mechanism, saving both time and memory. The interfaces are:

- st_malloc
- st_free

Which behave exactly like malloc and free, but while malloc and free get wrapped by the simulator to manage dynamic memory that gets rolled back, st_malloc and st_free do not, and allocate unmanaged memory.

Responding to events. The user needs to implement a series of functions that behave as interfaces for handling a series of events. These interfaces are:

- NeuronHandleSpike
- NeuronWake
- SynapseHandleSpike
- ProbeRead

NeuronHandleSpike is called by the framework when a spike event is delivered to the neuron. The module provides the intensity of the spike, and the model is asked to handle it by updating the state of the neuron and possibly scheduling a spike through SendSpike or MaybeSpike or MaybeSpikeAndWake.

NeuronWake is used to handle a neuron being woken up by the framework. At the moment, this only happens after a neuron fires because a valid MAYBESPIKEWAKE

event is received and processed. The purpose of waking up a neuron is to let it compute its next fire time (if any) and schedule the (hypothetical) future spike.

SynapseHandleSpike is used when sending spikes, signifying the transmission of the signal through the synapse. In this, the user needs to update the synapse state and compute the synapse weight, that has to be returned. This way, the user can have dynamic synapses with plasticity (either because of the state changing, or simply because of plasticity variation in time) and the module can properly include the intensity of the spike in the spike message. SynapseHandleSpike is invoked during a SendSpike, when iterating over all the synapses of the spiking neuron, and is invoked once per synapse.

ProbeRead is invoked after probed neurons spike. Its purpose it to let the user gather information on the neurons it has probed. No modifying the state or raising events should take place in this.

Management events. The last kind of interfaces need to be implemented by the user, and are:

- NeuronCanEnd
- GatherStatistics

NeuronCanEnd is invoked when the system performs the check for ending condition. While the simulation can be stopped after reaching a given virtual time, one might want to check on an ending condition. This is where it happens. NeuronCanEnd is only invoked on probed neurons.

GatherStatistics is invoked by the framework when completing the simulation. It lets the user know that the simulation is complete and that the time to output information to file has come, as it is now safe.

5.1.2 Data structures

Two main data structures are used by the module to properly transparentize the execution for the user. The data structures are:

- ___neuron_s, managing the entire LP state

- ___syn_t, which are used to keep the synapse information

The way the module behaves, it lives in a limbo between the simulator and user space: the module uses the SetState facility (used to communicate to the simulator where the current LP's state is in memory) to set its ___neuron_s struct as state, and catches the ProcessEvent calls—issued by the simulator when an event is delivered to a LP—to perform its magic.

The ___neuron_s struct contains: an array of (pointers to) ___syn_t objects, which are the neuron's outgoing synapses; a pointer to the user's neuron state; a pointer to an instance of mbspk_str, which is used to keep the identifier for the MAYBESPIKE/WAKE events; a probed flag. Since the physical composition of the neuron is supposed to be static, that is, the neuron does not change synapses and does not get "unprobed", the ___neuron_s struct is not monitored for rollbacks. However, the mbspk_str needs to be dynamic, as the identifier needs to evolve with the simulation, and possibly be rolled back when needed. This is why we use a pointer to dynamic memory, allocated and tracked by the memory manager, and transparently rolled back when the LP is too. The user's neuron obviously needs to be dynamic too and in ___neuron_s only the pointer to it is stored, as the framework completely ignores its nature. The memory area is allocated by the user during NeuronInit, and user's mallocs are automatically wrapped to monitored allocations.

___syn_t structs are more complicated than just synapses. One instance contains the ID of the synapse's target neuron, the transmission delay, and the pointer to the user synapse state, the nature of which, as with the neuron state, is completely ignored by the simulator. ___syn_t objects are also static throughout the simulation, as synapses may not change target or delay in the current implementation. A single ___syn_t object is created when NewSynapse is invoked and the target neuron and transmission delay are given by the user as parameters. As mentioned, NewSynapse acts as a malloc as far as the user is concerned, and the allocated memory area is pointed to from ___syn_t, whether the user synapse is dynamic or not, so that if it is, it gets rolled back properly.

5.1.3 The simulation flow

In our approach, the model has to be written as a C program which imports the simulator's header file, and implements the functions *NeuronInit*, *SNNInitTopology*, *NeuronHandleSpike*, *NeuronWake*, *SynapseHandleSpike*, *ProbeRead*, *NeuronCanEnd*, *GatherStatistics*. The model files then have to be compiled using the ad-hoc compiler, which outputs two executables: model_serial and model_parallel. When one of the executables is launched, the number of LPs has to be specified as command line argument, and then the simulation is started. The user is also allowed to specify their own custom command line arguments and handlers through the argp library, letting them configure the simulation run.

From the user's perspective, the first implemented function (apart from the argp handlers) that is executed is NeuronInit. Neurons are indeed mallocd and initialized before the topology is. Then SNNInitTopology is invoked, the topology is initialized, synapses are created, neurons are probed, and spike trains scheduled. Then the simulation starts, and the handlers are called by the framework. The first call happens at virtual time 0 and is a spike with intensity zero. This is used to wake up neurons and allow self-spiking ones to compute and schedule their next fire times. After this the simulation proceeds until one of the ending conditions are met: there are no more spikes left to process, or the maximum simulation time has been reached, or the ending predicate is verified.

From the point of view of the framework, the call to module initialization is hooked to be performed immediately after the initialization of the memory manager and the libraries. The call is performed on every thread. As such, attention is paid to actually initializing exclusively objects that belong on LPs managed by the local thread. The module initialization function computes how many LPs the thread manages and the ID of the first of them, then sets up the facilities to manage the neurons (___neuron_s objects) in each of the owned LPs, and calls the NeuronInit for every neuron so as to let the user malloc and initialize said neuron and obtain pointer to the user's neuron state. Then the active LP is set to the first managed by current thread, and (the user-defined) SNNInitTopology is called, and in it the user may invoke NewSynapse, NewProbe, and NewSpikeTrain. Being that the execution

is multi-threaded, but these functions have side effects on data structures hosted on specific LPs, each of these functions first checks whether the interested LP is under the responsibility of the current thread, otherwise the operation falls through. After the user returns from SNNInitTopology, the duplicated synapses are pruned, and synapse arrays are compacted, all to recover used memory that would otherwise be wasted. Then control is given back to the simulator, that starts the run.

The first raised event is INIT at virtual time 0, which is caught by the module, that delivers the above-mentioned spike with intensity 0 to all the neurons to let them schedule spikes. From this moment, the simulation proceeds based on the events that are raised.

Chapter 6

Experimental Assessment

The experimental assessment has been conducted in two phases: first, correctness was verified, then performance was evaluated.

In doing so, a Leaky Integrate and Fire neuron model to be used as building block of the test networks has been implemented. The physical parameters were taken from T. C. Potjans and M. Diesmann's work [13]. The network reported in the aforementioned work ([13]) has been used in our testing as the reference model. Other networks have been completely made up to verify correctness.

6.1 The neuron implementation

In this section, the two kinds of neurons that were implemented are reported.

6.1.1 Leaky Integrate and Fire

The Leaky Integrate and Fire neuron model was introduced earlier, in Section 2.3. The physical parameters used and their meanings are reported in Table 6.1 under the *Neuron Model* part. The equations have been taken directly from the code of [15]—a more recent reimplementation of the Potjans and Diesmann's work on the Brian simulator—, which reported them in string format (as Brian takes the model's differential equations in string form). Note that the neurons take into account the refractory period of duration τ_{ref}, and as such the membrane voltage is clamped to V_r for the entirety of the refractory period after spiking.

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{-V(t) + V_r}{\tau_m} + \frac{I(t) + I_{ext}}{C_m} \tag{6.1}$$

$$\frac{\mathrm{d}I(t)}{\mathrm{d}t} = \frac{-I(t)}{\tau_{syn}} \tag{6.2}$$

Equations 6.1 and 6.2 are the ones reported that describe the LIF neuron model. Note that Equation 6.1 is simply a rewriting of Equation 2.6 in Section 2.3, where τ_m has been brought to the right of the equals sign, and $I(t)$ has been expanded into $I(t) + I_{ext}$ where $I(t)$ is now the current coming from incoming synapses, and I_{ext} is the current that raises as a consequence of the background input. I_{ext} has also been used to verify the correctness of the model in the two custom networks, by simulating input spike trains thanks to neurons that were made self-spiking because of I_{ext}.

Both Equation 6.1 and 6.2 were solved in closed form for faster and more precise computation, yielding Equations 6.3 and 6.6.

$$V_f = e^{-\frac{\Delta t}{\tau_{syn}}} I_i A_0 + A_2 + (V_i - A_2 - I_i A_0) e^{-\frac{\Delta t}{\tau_m}} \tag{6.3}$$

$$A_0 = \frac{1}{\left(\frac{1}{\tau_m} - \frac{1}{\tau_{syn}}\right) C_m} \tag{6.4}$$

$$A_2 = \left(\frac{V_r}{\tau_m} + \frac{I_e}{C_m}\right) C_m \tag{6.5}$$

$$I_f = e^{-\frac{\Delta t}{\tau_{syn}}} I_i \tag{6.6}$$

In the equations, the two members A_0 and A_2 have been introduced for the sole purpose of readability and their expansions are found in Equations 6.4 and 6.5. However, for the implementation, it is worth noting that A_0 and A_2 are constants based on the neuron's physical parameters, and A_2 is the value at which the membrane potential tends to when left with no inputs apart from $I_e xt$. As such, in our implementation they are computed once during the initialization and stored along the other physical parameters of the neuron.

Neuron parameters and state

Global parameters. The model needs to have its own state for the neurons. In our case, since all physical parameters are shared between all neurons in the model, they have been stored in a single global struct allocated on static memory that is accessed from all the workers, posing no danger as it is accessed as read-only. This struct is n_params and contains V_{th}, V_r, $\frac{1}{\tau_m}$, $\frac{1}{\tau_{syn}}$, $\frac{1}{C_m}$, τ_{ref}. The inverses of τ_m, τ_{syn}, and C_m are stored because they always appear in the denominator, and multiplying by the inverse is way faster than dividing. There is only one calculation during the run in which the model actually needs to multiply by τ_{syn}, in which we divide by its inverse.

Population-level parameters. Then there are population-level parameters, that all neurons in a given population share. These parameters are computed at run time and are stored in neuron_helper_t structs, of which there are 8, one struct per population, visible to the whole thread and allocated on non-rolled-back memory. Since they are computed in a deterministic manner, there would be no race conditions even if the struct was global even across threads, since even in case of a double write access, the data written would be the same. The neuron_helper_t structs contain: I_{ext}, A_0, A_2, I_{cond}, $self\_spike\_time$. The last two parameters are mentioned here for the first time. I_{cond} is defined in Equation 6.7.

$$I_{cond} := I(t) \mid \lim_{V(t)\to V_{th}} \frac{\mathrm{d}V}{\mathrm{d}t} = 0 \tag{6.7}$$

An $I(t) > I_{cond}$ is needed to overcome the threshold value and spike, when $V(t) \to V_{th}$. This means that having $I(t) > I_{cond}$ is the **necessary condition** for a non-self-spiking neuron to spike in the future (for some $t' > t$). Self-spiking neurons will have an $I_{cond} < 0$. Thus, I_{cond} has two functions: first, if $I_{cond} < 0$ we know that a neuron is self-spiking; second—and most importantly—, it gives us a sufficient condition to say that the neuron will not spike in the future: a non-self-spiking neuron that has $I(now) < I_{cond}$ will surely not spike in the future, as $I(t)$ naturally decays towards 0; as such, the necessary condition for firing will never be met in the future (unless a new spike is received, but that is for a future event). Another

extremely important aspect that I_{cond} is useful for is computing an upper bound for the Δt after which the neuron will spike: once we have checked that, in a non-self-spiking neuron, $I(now) > I_{cond}$, we can compute $\Delta t \mid I(now + \Delta t) = I_{cond}$. We know that this Δt surely exists because $I(t)$ is decaying towards 0 and $I_{cond} > 0$. We can compute Δt simply by solving Equation 6.6 for t, posing $I_f = I_{cond}$. We obtain Equation 6.8.

$$\Delta t = -\ln\left(\frac{I_{cond}}{I(now)}\right)\tau_{syn} \tag{6.8}$$

Once we have Δt we have an upper bound to the spike time, and we can even determine whether a neuron spikes or not in the given interval of time. We have two cases: either $V(now + \Delta t) \geq V_{th}$, or not. If we are in the first case, then the neuron spikes at a time $t_{spk} \in [now, now + \Delta t]$, otherwise the neuron does not spike. Indeed, since I_{cond} is needed to overcome V_{th} with $V(t)$, we also know that any $I(t) > I_{cond}$ is sufficient to **keep** $V(t) > V_{th}$. As such, if there is a time $t^* \in [now, now + \Delta t]$ for which $V(t^*) > V_{th}$, this will stay true for at least as long as $I(t) > I_{cond}$, i.e. for all $t \in [t^*, now + \Delta t]$. Thus, if $V(now + \Delta t) \geq V_{th}$, V_{th} is crossed **once** in $[now, now + \Delta t]$, and we have that $t_{spk} = min(\{t^* \in [now, now + \Delta t] \mid V(t^*) > V_{th}\})$. Otherwise, we never cross V_{th} and neuron does not spike.

Once Δt is computed, we check whether $V(now + \Delta t) \geq V_{th}$, and if so, a binary search is conducted in the interval $[now, now + \Delta t]$ to find the spike time t_{spk}.

The other parameter, self_spike_time, indicates how long a self-spiking neuron takes to spike in total absence of synaptic inputs (i.e. $I(t) = 0$). It is computed upon initializing a population's n_params struct if $A_2 > V_{th}$ i.e. if the neurons of the population are self-spiking (checking that $I_{cond} < 0$ would have been equally valid). self_spike_time is useful for computing the next spike time of self-spiking neurons: while non-self-spiking neurons can exploit the procedure explained above, self-spiking neurons cannot. As such, self_spike_time is used as a step to move the time window to find the spike time for self-spiking neurons: $I(t)$ decays towards 0 whether it is positive or negative. If positive, then the spike time will come faster, and will more likely be contained in the interval $[now, now + self\_spike\_time]$, if negative, it will take longer. In both cases, if $V(now + self\_spike\_time) < V_{th}$, the interval window

is slid to the right by self_spike_time, becoming $[now + self\_spike\_time, now + 2self\_spike\_time]$ and so on, until $V(now + k * self\_spike\_time) \geq V_{th}$, at which point the search will take place in the interval $[now + (k-1)self\_spike\_time, now + k * self\_spike\_time]$. In a nutshell, self_spike_time represents a reasonable step size to conduct the search for the spike time, when no upper bound is available.

Neuron state. Finally, the neuron state holds the dynamic variables of the neuron, and as such is allocated on dynamic, monitored memory: *times_fired*, *membrane_potential*, *I*, *last_fired*, *last_updated*. The field membrane_potential holds $V(last\_updated)$, and I holds $I(last\_updated)$. The other fields are self-explanatory. An additional last field is *helper*, which holds a pointer to the neuron_helper_t struct of the population the neuron belongs to. This pointer is needed to know where the population-level parameters are, and is necessarily dynamic even if unmonitored memory would have been enough: a small price to pay.

Evolving through time

The neuron's evolution through time is dictated by two main functions: bringToPresent and getNextFireTime.

bringToPresent takes care of physically updating the neuron state from a past time, to the *now* time t_{now}. It is the first function called both from NeuronHandleSpike and NeuronWake. It first checks whether the neuron is in refractory period. If it is, the function handles it by computing when it will end: if it ends after (or at) the current virtual time t_{now}, then $V(t)$ is set to V_r, $I(t)$ is updated using Equation 6.6, using the difference between t_{now} and last_updated as Δt; then last_updated is set to t_{now}, and the function ends. If the refractory period ends at a time $t_{r\_end} \in [last\_updated, t_{now})$, then the update takes place in two phases: from last_updated to t_{r_end}, and then from t_{r_end} to t_{now}. Thus, first the refractory period is handled, setting $V(t)$ to V_r, and updating $I(t)$ using Equation 6.6 with Δt the difference between the t_{r_end} and last_updated, then the evolution is computed again normally, updating $V(t)$ using Equation 6.3 and $I(t)$ using Equation 6.6, both with $\Delta t = t_{now} - t_{r\_end}$, then last_updated is set to t_{now}, and the function ends. If

instead the neuron is not in refractory period, the evolution is computed by simply using the correct equations to compute $V(t_{now})$ and $I(t_{now})$, last_updated is set to t_{now}, and the function ends. It is immediately noticed that the function does not contemplate the possibility of spiking, this is because this would mean scheduling an event in the past, which is impossible. Scheduling of spikes is taken care of when waking up or handling a spike, which schedule spikes for the future. As such, there cannot be unscheduled past spikes.

Between bringToPresent and getNextFireTime, the effects of the spike are applied to the neuron: $I(t)$ is incremented by the intensity of the spike. Then getNextFireTime is called.

getNextFireTime is the function that, under the assumption of receiving no spikes in the future, computes whether and when the next spike will happen. It is the second function called in NeuronHandleSpike and in NeuronWake. Please note that the role of I_{cond} as explained in 6.1.1-*Population Level Parameters* is vital to this explanation. First of all, the function checks whether the neuron is self-spiking, and if it is not, it checks whether $I(t_{now}) < I_{cond}$: if it is the case, the neuron will never spike and the function returns. If the neuron is self-spiking or $I(t_{now}) > I_{cond}$, then the function moves on to check whether the neuron is still in refractory period: if it is, V is set to V_r, and $I(t_{r\_end})$ is computed according to the closed form equation. Once the refractory period is out of the way, the paths of neurons and self-spiking neurons separate: if the neuron is self-spiking, the spike time is found using a function that uses self_spike_time as a step to look into the future, as mentioned above. For non-self-spiking neurons, it is checked once again if $I(t_{r\_end}) < I_{cond}$, and once again if this happens, the neuron will never spike and the function ends. Otherwise, the spiking time upper bound $t_{upper} = max(t_{now}, t_{r\_end}) + \Delta t$ is computed, using Equation 6.8 to compute Δt. Then $V(t_{upper})$ is computed using Equation 6.3, and—because of what was explained earlier when introducing I_{cond}—it checks whether $V(t_{upper}) < V_{th}$: if so, the neuron will never spike and the function returns. However, if $V(t_{upper}) \geq V_{th}$, then we know (again, because of the reasoning in 6.1.1) that the neuron spikes at some time $t_{spk} \in [t_{now}, t_{upper}]$. Now that we have an the interval of time in which we know a spike has happened, binary search is employed to find t_{spk}

and return it.

Once the fire time of the next spike is known (if any), a hypothetical spike is scheduled through MaybeSpikeAndWake for the computed time.

6.1.2 Poisson neurons

Poisson neurons were implemented because they were used as thalamic input to Potjans and Diesmann's network. Poisson neurons are simply neurons that spike according to a Poisson distribution, given a fixed spiking rate. In our implementation they have no state, as the spiking rate is the same for all Poisson neurons and globally defined. The function implemented to emulate a Poisson neuron is *PoissonWake*: when *NeuronHandleSpike* or *NeuronWake* are called with target a Poisson neuron, PoissonWake is called. Since Poisson neurons cannot receive input, the only time NeuronHandleSpike is invoked with target a Poisson neuron, is at time 0 when starting the simulation. One needs to make sure Poisson neurons are never the targets of another neuron's spike, otherwise their behaviour would be compromised. Inside PoissonWake, the interval Δt after which the neuron will spike is computed by extracting a value from an exponential distribution with rate $\frac{1}{\lambda_{thal}}$.

Since in the Potjans and Diesmann's model thalamic input was transient, with only 10 milliseconds of activity time every second, this was embedded in the PoissonWake function, which also takes care of computing the correct time to schedule the spike, knowing that for each second, the neuron is asleep for 700ms, then awake form 10ms, and again asleep for 290ms: as an example, if $\Delta t = 82$ (with Virtual Time in milliseconds), then we have 8 skipped cycles (for a total of 80ms of activity) and then 2 more milliseconds of activity to be waited; if we are at time 0, then the spike will be scheduled at $t_{spk} = 8702$. Spike scheduling happens via *MaybeSpikeAndWake* that allows to wake up the neuron to schedule the next spike.

6.2 The networks

In this section, the networks used to test our work are introduced.

6.2.1 Potjans and Diesmann's Local Cortical Microcircuit

Potjans and Diesmann (in [13], 2014) developed a model of the local cortical microcircuit. They did so by using extensive experimental data on the circuitry of striate cortex. At the time of their research, there were a great deal of Local Cortical Microcircuit models, but there was a mismatch between simulated and activity measured in-vivo. The research mostly focused on developing networks with different types of neurons. However, simply changing the neuron implementations with more complex ones yielded no improvement. As such, they believed the error was in the adopted connectivity map, rather than in the neuron models. The two scientists set out to create a connectivity map that "integrates the 2 major connectivity maps from anatomy and electrophysiology and furthermore incorporates insights from photostimulation and electron microscopy studies", and did so by algorithmically combining the two maps "by correcting for the different experimental procedures".

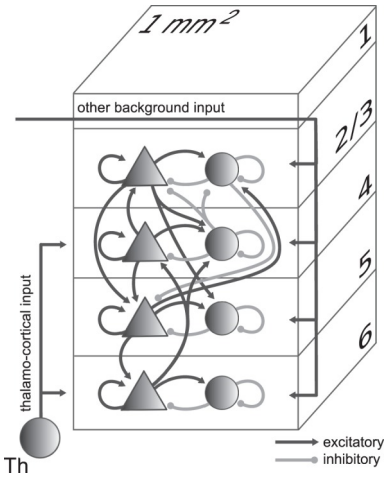


Figure 6.1. Potjans and Diesmann's Model definition. Excitatory populations are represented by triangles, and inhibitory populations are circles.
Source: [13]

The network Potjans and Diesmann came up with comprises 4 layers of cortex, named 2/3, 4, 5, and 6, each with an excitatory and an inhibitory neuron population. Layers are connected to one another according to the "*connection probability* of a connection" that "defines the probability that a neuron in the presynaptic population forms at least 1 synapse with a neuron in the postsynaptic population. A *connectivity map* is defined by the 64 connection probabilities between the 8 considered cell types".

Table 6.1. 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

The synapses in the model are static. The number of neurons per population are chosen according to [4]. Furthermore, every layer can receive a background input in the form of a continuous current, and input from an external thalamo-cortical neuron population. The thalamic neurons are implemented as Poisson neurons with a fixed spiking rate. In Figure 6.1 is the very representation of the network that Potjans and Diesmann drew.

Table 6.1 contains the neuron parameters: population size for each population, external inputs for e each population, and the neuron's physical properties. Table 6.2 contains the connectivity map and the synaptic connection parameters.

In their work, Potjans and Diesmann found that when simulated, the model using their integrated connectivity map yielded "cell-type specific spontaneous and stimulus-evoked activity in good agreement with experimentally observed activity". Indeed their model is still extremely relevant, with its implementation being part of the examples in the NEST simulator, and has been replicated in 2018 by Shimoura et al. [15] in an implementation for the Brian simulator.

It is this latter implementation the one against which we tested our work. Our

Table 6.2. 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

model was implemented by replicating the topology Potjans and Diesmann proposed, tracing the Python code that Shimoura et al. wrote for Brian, as far as initializing the network goes.

6.2.2 Other networks

Two more made up networks have been implemented for testing. The first is a network with 2 neurons, N_0 and N_1, which we will call **Test Network A**. N_0 receives a constant input current, and N_1 receives the output of N_0 as input. The purpose of this network was to test correctness of our implementation and see how precisely the behaviours in the different simulators matched.

Table 6.3. Connectivity map for Test Network B.

		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

The second network, *Test Network B*, is a network made of 1000 Leaky Integrate and Fire neurons, which is 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 LIF neurons, each of which receives a constant current input, and each of the three layers has 200 excitatory and 100 inhibitory LIF neurons. It employs a connectivity map (in Table 6.3) to connect the populations with each other. Its purpose was simply to model a smaller network that was faster to simulate, and then compare the data for correctness.

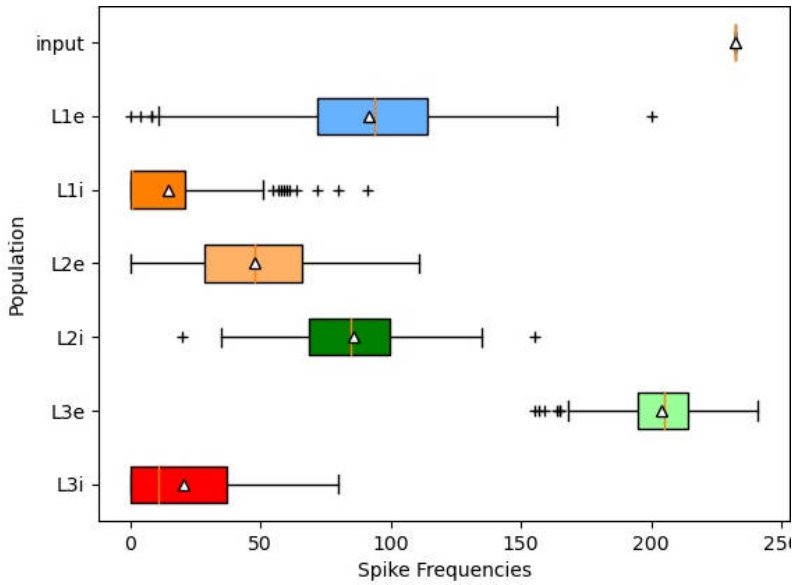


Figure 6.2. Spiking rates (Hz) for 1000 neurons model on our framework.

6.3 Correctness

To validate the framework's behaviour and model implementation, the aforementioned models have been executed both on our simulator and Brian.

The two neuron model was used to validate the implementation of the single neuron. The supply of the constant input current to N_0 was simulated by setting N_0's I_{ext} to 1800 pA and synaptic weight to 200 pA, to allow N_1 to spike using solely the input from N_0. Then the network was run and the results were compared: with all the tested configurations, the neurons spiked about 1% less than their counterpart when running on our simulator compared to Brian. This is likely because of the error introduced by the approximate resolution of the differential equation in Brian. This result shows that the implementation of the neurons is correct, both when receiving input through I_{ext} and when receiving spikes from other neurons, and the implementation of the framework is also correct when input spikes are in order.

The 1000 neuron model was used to validate ensemble behaviour of the network and the behaviour on differently timed spikes. Once again, the same network was run on our implementation and on Brian. A boxplot of spiking frequencies of every population is reported. As a reminder, the 1000 neuron model was split into one input population of 100 neurons and 3 layers, each with one excitatory population

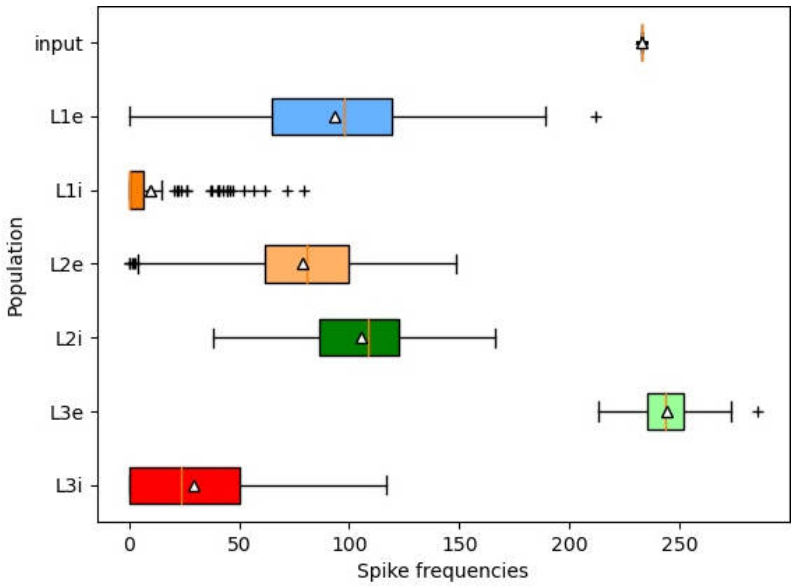


Figure 6.3. Spiking rates (Hz) for 1000 neurons model on Brian.

of 200 neurons and one inhibitory population of 100 neurons. The boxplots of the executions are reported in Figure 6.2 and 6.3. As we can see from the figure, the boxplots of spiking frequencies closely resemble one another. The difference is likely a consequence of the randomized initialization of the network topology, with the small size of the network not being enough to amortize randomness, along with the 1% difference introduced by precision. This shows how the framework and LIF model implementation of the neurons are correct even when a greater number of neurons is simulated, and with neurons handling spikes coming in varied patterns from many other neurons.

Finally, Potjans and Diesmann's model was used to test Poissonian neuron behaviour, and most importantly behaviour on a large scale. Furthermore as it is the model used to compare performance, it was of vital importance to make sure that the model was correct. Once again, the model has been run on both our simulator, and the result compared with that of Shimoura et al. and Potjans and Diesmann's using boxplots of spiking rates. The plots for our simulation is found in Figure 6.4, while that of Shimoura et al., obtained by running on Brian, is found in Figure 6.5. Lastly, the boxplot of the original paper, obtained by running on NEST, is in Figure 6.6.

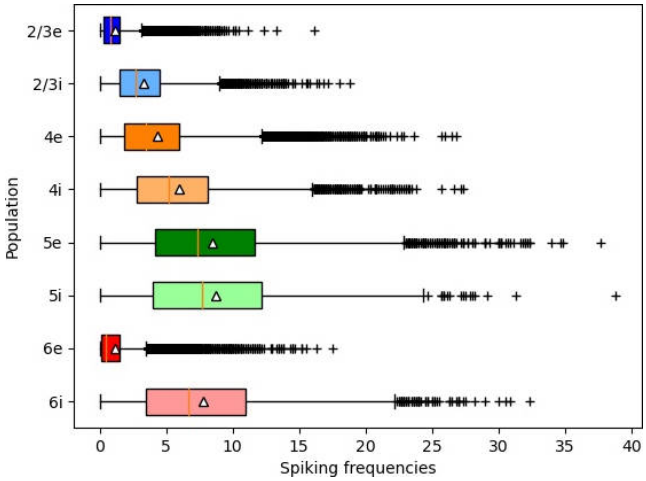


Figure 6.4. Spiking rates (Hz) for Potjans and Diesmann's model on our framework.

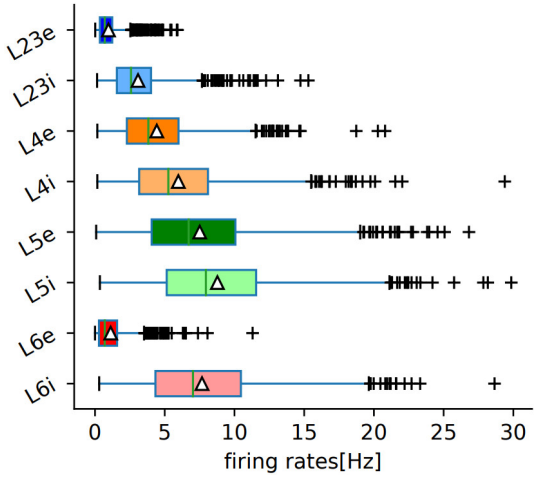


Figure 6.5. Spiking rates (Hz) for Potjans and Diesmann's model on Brian. Source: [15]

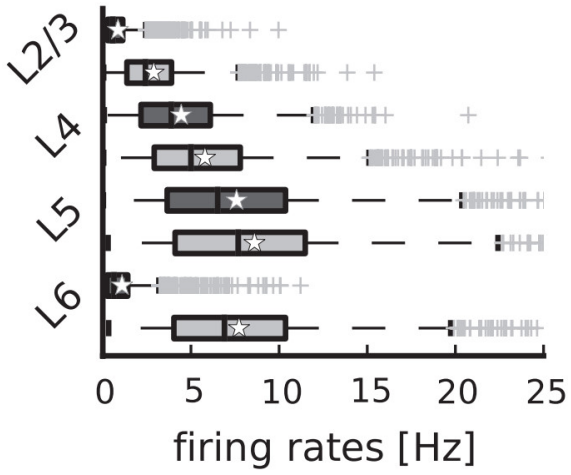


Figure 6.6. Spiking rates (Hz) for Potjans and Diesmann's model on NEST. Source: [13]

The boxplots once again reassure us that the model implementation is correct, and yields results coherent with both implementations of the model. Poissonian neurons are also implemented correctly.

6.4 Performance

Performance was evaluated by simulating our implementation of Potjans and Diesmann's Local Cortical Microcircuit model. Tests were carried out on a NUMA machine with 4 AMD Opteron 6168 CPUs with 12 cores each, for a total of 48 cores and 8 NUMA nodes, with a total of 126GB RAM. When running the model, the I_{ext} was normalized for each population based on its member count. Two tests were executed with our approach to gather run times and speedups: 10 seconds runs and then 20 seconds runs, executed with worker thread counts (and, as such, core counts) multiples of 3, starting from 3 all the way to 48. The variation in Wall Clock run time when changing the number of worker threads has been plotted and is presented in Figures 6.7 and 6.8, for 10 and 20 seconds of simulation time respectively. A speedup graph could not be plotted because of the enormous time that running this computation with only one worker thread would have required, which made timing the execution with a single core impossible in reasonable amounts of time. The speedup can nonetheless be easily deduced from the presented graphs: we are in presence of a superlinear speedup up until 9 worker threads are used. This is likely because of the Operating System distributing threads onto different NUMA nodes, which generates less contention on the RAM bus. After 9 threads the speedup continues, but is then linear or sub-linear: the proposed approach exhibits

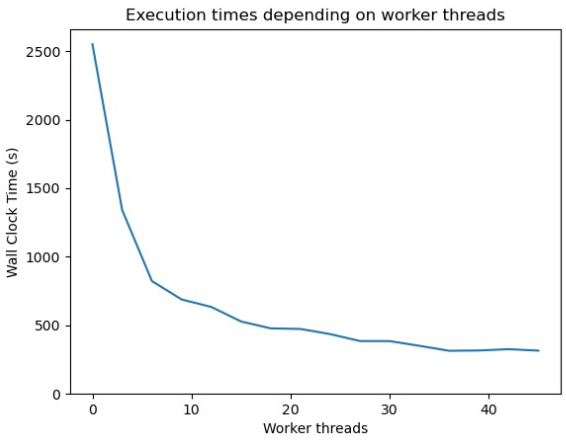
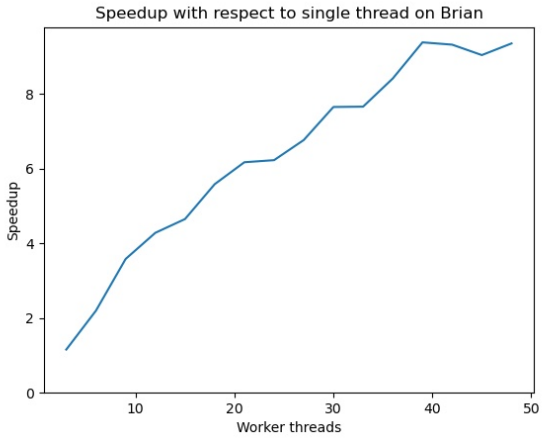


Figure 6.7. WCT elapsed to simulate 10 seconds, in relation to the worker threads.



ht

Figure 6.9. Speedup with respect to single-threaded on Brian.

weak scaling. This behaviour is even more evident in Figure 6.8, in which the running time is much higher for lower numbers of worker threads, but rapidly falls off to amounts easily comparable to those of Figure 6.7 when using higher thread counts. It is also worth emphasizing that since the network initialization is parallel too, the part of the computation that is not parallelizable is so minuscule that its effects on scalability are negligible.

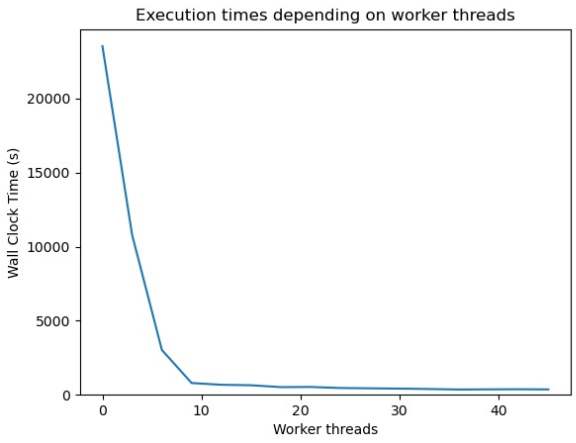
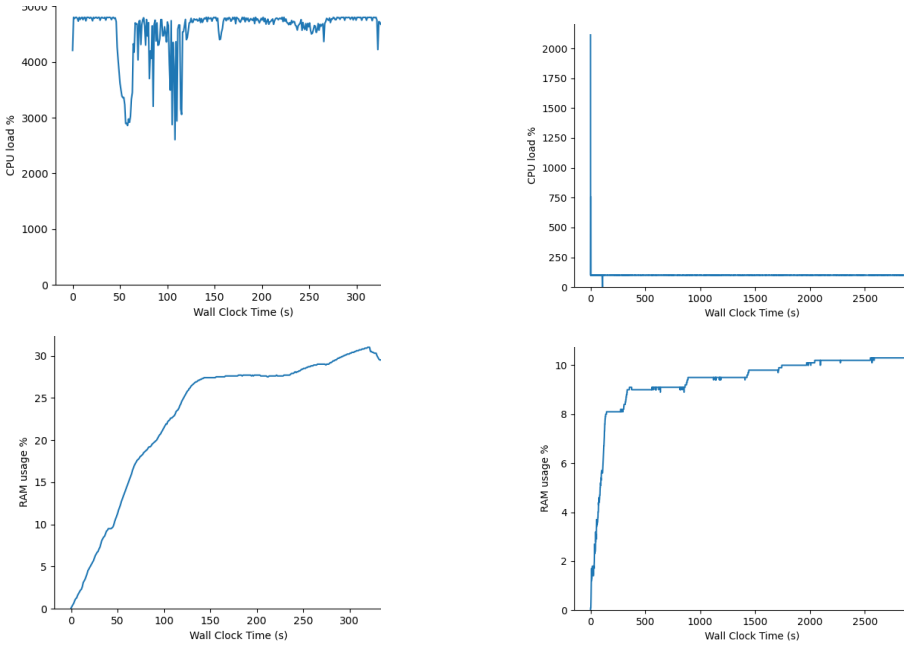


Figure 6.8. WCT elapsed to simulate 20 seconds, in relation to the worker threads.

The performance was compared with that of Shimoura et al.'s [15] implementation of Potjans and Diesmann's model, which is based on Brian. The simulation was run to 10 seconds virtual time on a single core as, notwithstanding the claimed OpenMP capabilities, execution reverted to single core even when instructed to run with multiple threads. This is probably because Brian's ability to **transparently** render code multi-threaded depends on the possibility of running



(a) CPU and RAM usage for 48 worker threads (b) CPU and RAM usage for Brian single thread execution

Figure 6.10. Plots of CPU% and RAM% usage for our framework and Brian.

it in its cpp_standalone mode. As such, Brian's performance can be used as a baseline to compute the speedup of our approach. The resulting graph is presented in Figure 6.9. We can see how, while Brian is faster on single thread, our approach consistently speeds up with respect to Brian's performance, when increasing the worker thread count.

CPU usage and RAM usage were also monitored during the various runs, and have been plotted. The plots can be found in Appendix A, while in Figure 6.10 only the plots of CPU and memory usage from the run with 48 worker threads are reported, alongside the plots of Brian's CPU and memory usage.

From the various plots we can see how in our approach the memory usage stabilizes after an initial phase. The end of the network initialization phase can be recognized thanks to the dip in CPU usage, as all cores synchronize before starting the simulation: it is the only synchronization point in the whole run. During network initialization RAM usage grows rapidly, as the various data structures supporting the execution are initialized and populated; the topology alone is fairly large, easily occupying more than 15GB of memory. Then at the same point in which the

first, bigger dip in CPU usage is found, RAM usage slows down: in this phase the simulation has started and the—slower, albeit still steep—rise in memory usage is the consequence of memory allocation for messages. During this phase, CPU usage can be seen dropping, as threads are waiting on the RAM. Once this other "message allocation" phase, RAM usage stabilizes and does not grow much for the rest of the run.

One could argue that RAM usage is still around 3x that of Brian, however this is the price to pay to achieve scalability thanks to Parallel Discrete Event Simulation and Time Warp: both state checkpoints and messages consume their share of memory. It should also be noted, however, that the current implementation was built in a very limited amount of time for such a complex application, which means that there is room for improvement. Improving RAM usage would also mean faster memory operations, leading to shorter run times. Furthermore, this kind of RAM usage is still very manageable, especially for HPC systems, that are the main target of this work: the huge speed boost brought about by the ability to heavily parallelize execution through optimistic synchronization, all the while hiding the underlying inherent complexity of the runtime, is the real objective and advancement sought in this work.

Chapter 7

Conclusions

In this dissertation, the path towards making high performance simulation of spiking neural networks accessible to all researchers was discussed, and our work in which promising steps have been taken to make it a reality, presented.

Thanks to this work, a modeller that knows nothing about Parallel Discrete Event Simulation can embark on the task of developing a Spiking Neural Network model for PDES and manage to achieve great performance, while only ending up knowing "something" about PDES, if at all. In particular, thanks to the APIs presented in Chapter 5, the user is guided through the development of the model in a way that forces it to be heavily parallelizable, achieving promising performances akin to those seen in Chapter 6, with weak scaling.

We have seen how the sheer complexity of simulating a continuous-time system such as SNNs using PDES can be masked through "hypothetical" events that are only executed if some condition holds (specifically, if no other event was delivered to the LP between the scheduling and handling of one such event), leading to easier handling of time continuity by the model.

Now what are the next steps to take? Firstly, while the RAM usage is already acceptable, using techniques to further reduce the memory footprint is key to improving processing speed. One example could be an ad-hoc memory manager for the kind of allocations that SNNs perform, which are usually small in size and in huge numbers. Another aspect that can see improvements is the event queues: while currently heaps are used, a properly configured calendar queue could easily

outperform heaps.

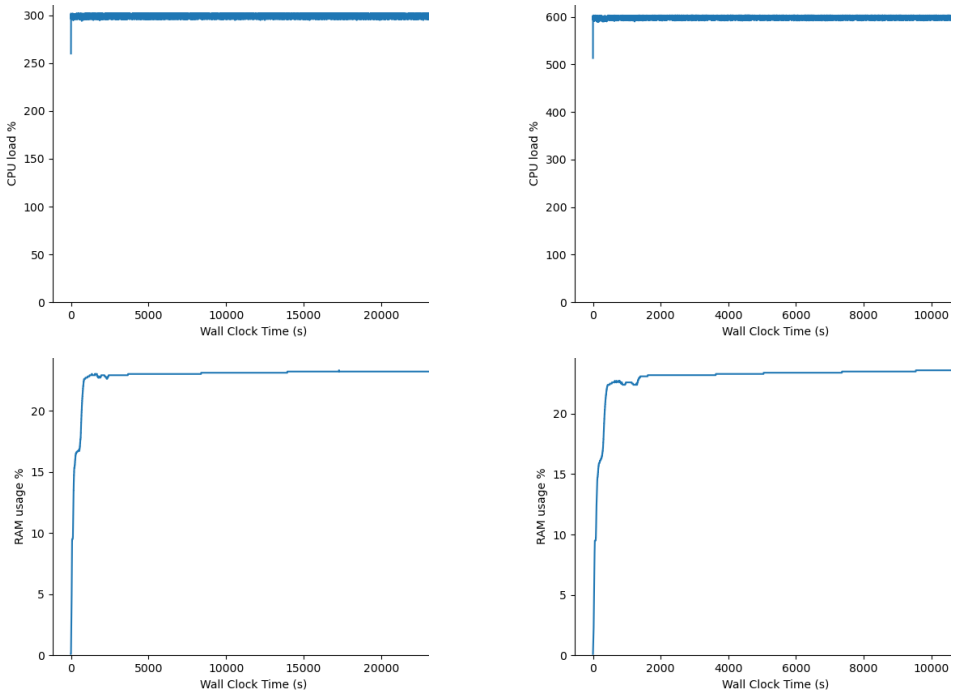
Another important aspect might be that of, like Brian, asking the user for just the differential equations representing the model, and simulating from that. This is however a complex aspect that would probably need its own work.

More aspects regard the management of neurons as populations: while currently neurons are treated as LPs, many—possibly related—neurons could be clustered and managed as a single LP. With the adoption of proper load balancing this might lead to better performance. This however should be further explored. Similarly, both before and at run-time, populations made of neurons that interact more closely with one another could be identified and be grouped together, which would further reduce the incidence of causality violations both in multi-thread and on multinode systems.

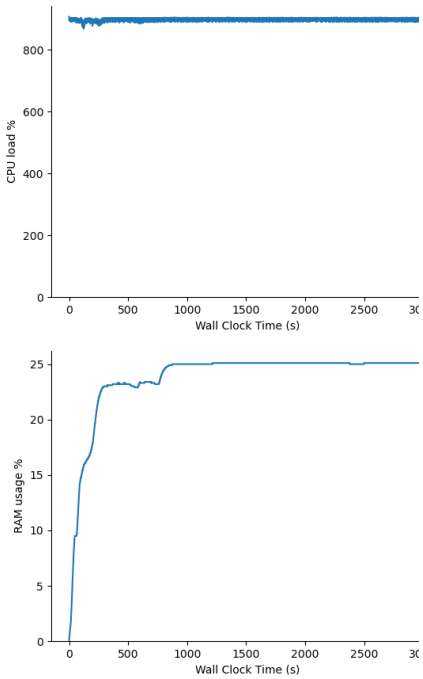
This work represents but a tiny step towards the greater goal of High Performance Simulation of Spiking Neural Networks, but will hopefully be one of the many small steps in the right direction.

Appendix A

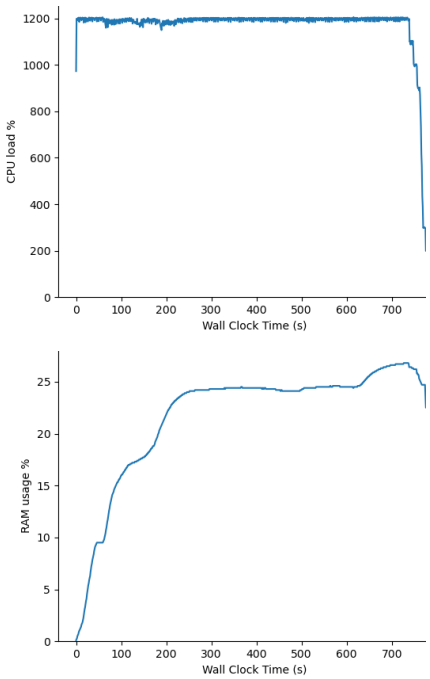
CPU and Memory Footprint



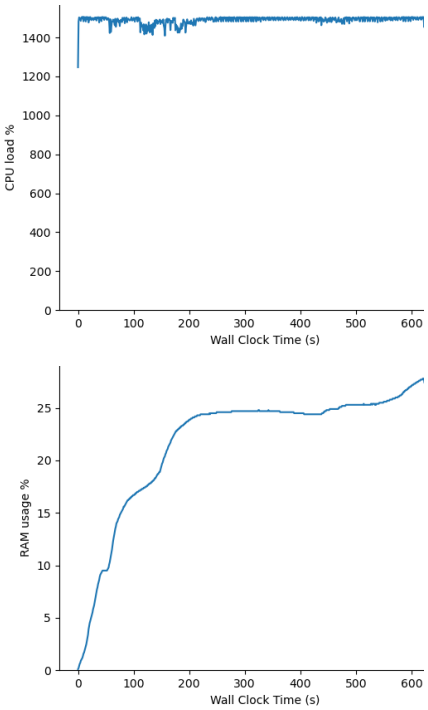
(a) CPU and RAM usage for 3 worker threads (b) CPU and RAM usage for 6 worker threads



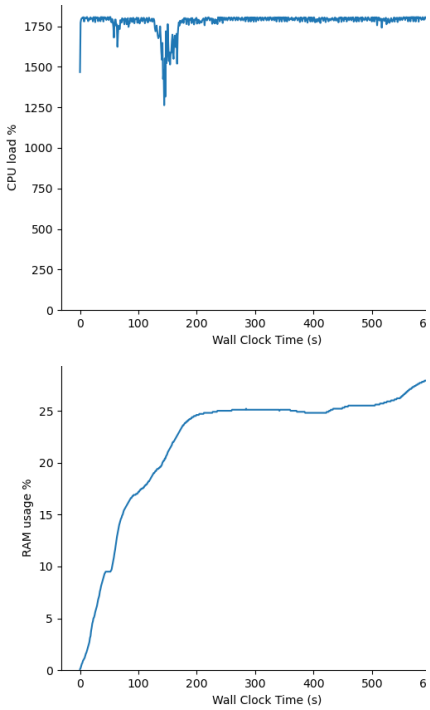
(c) CPU and RAM usage for 9 worker threads



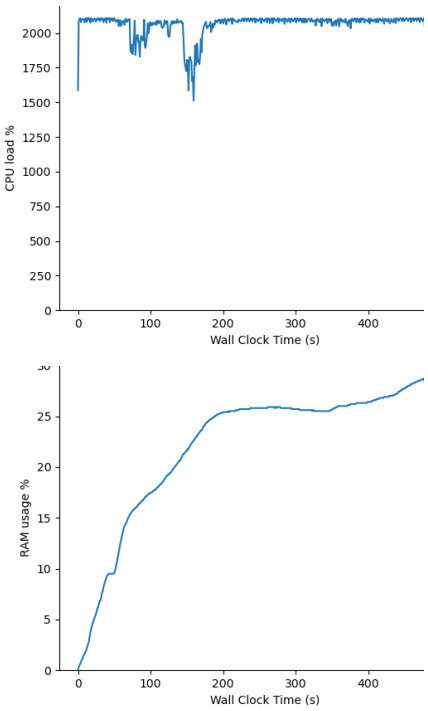
(d) CPU and RAM usage for 12 worker threads



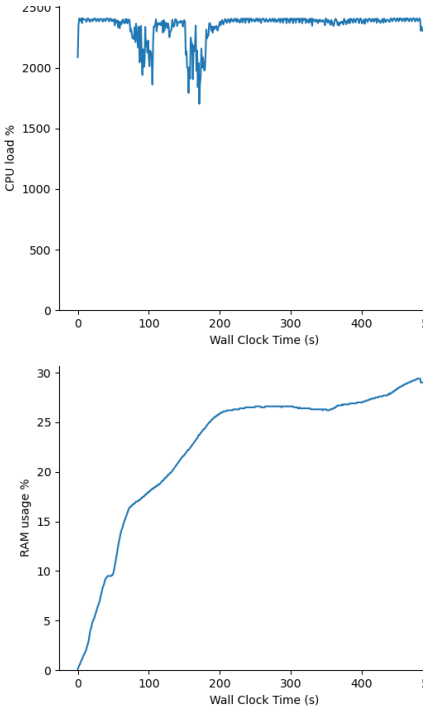
(e) CPU and RAM usage for 15 worker threads



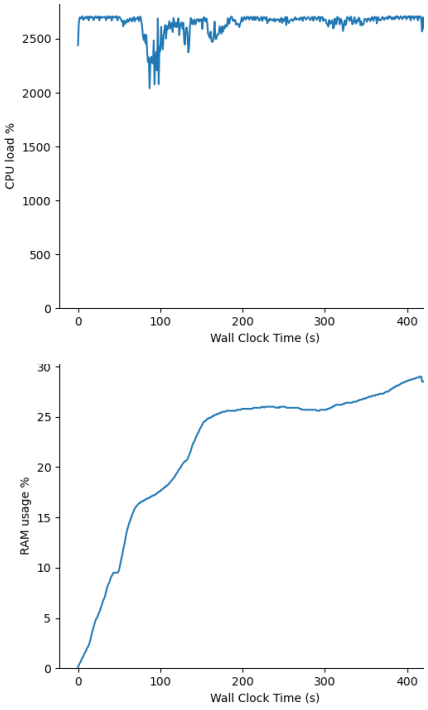
(f) CPU and RAM usage for 18 worker threads



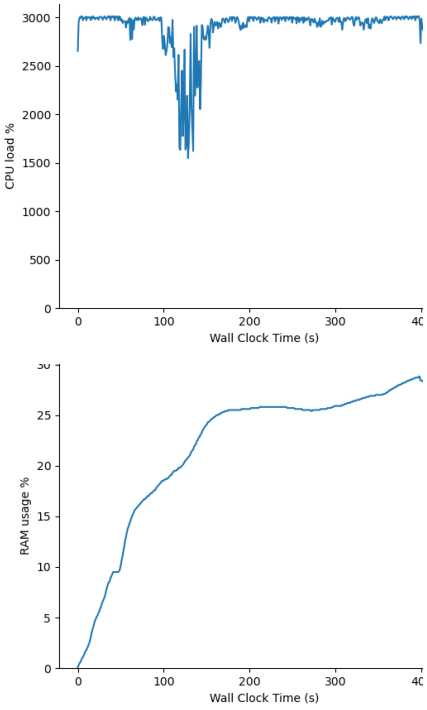
(g) CPU and RAM usage for 21 worker threads



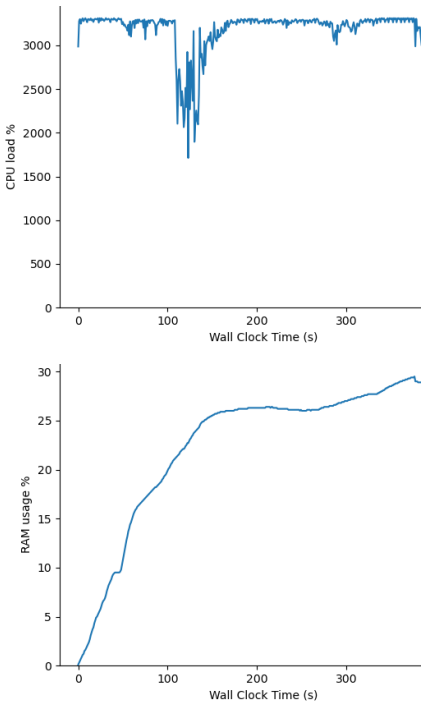
(h) CPU and RAM usage for 24 worker threads



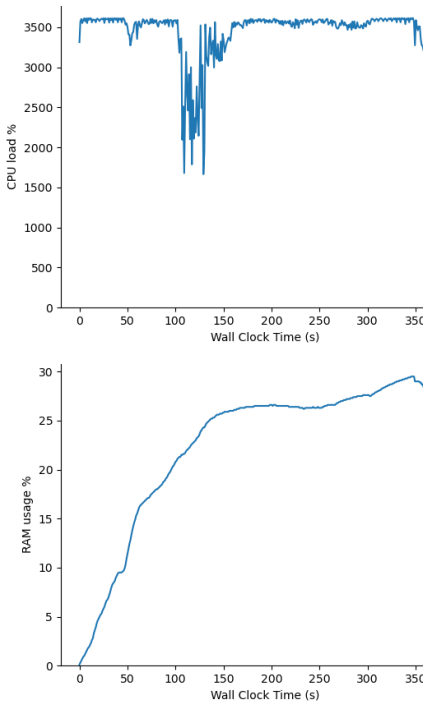
(i) CPU and RAM usage for 27 worker threads



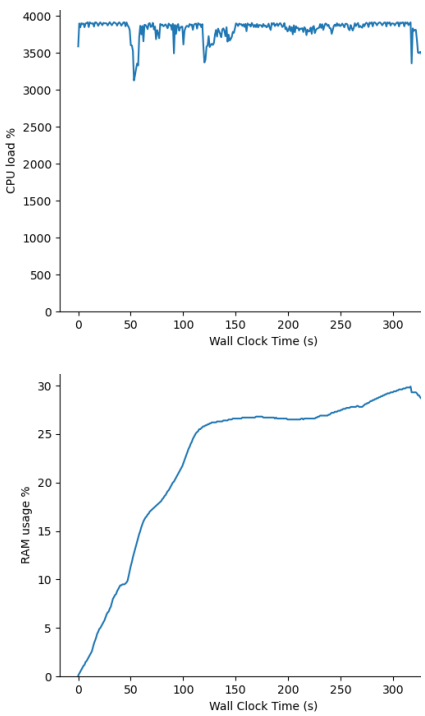
(j) CPU and RAM usage for 30 worker threads



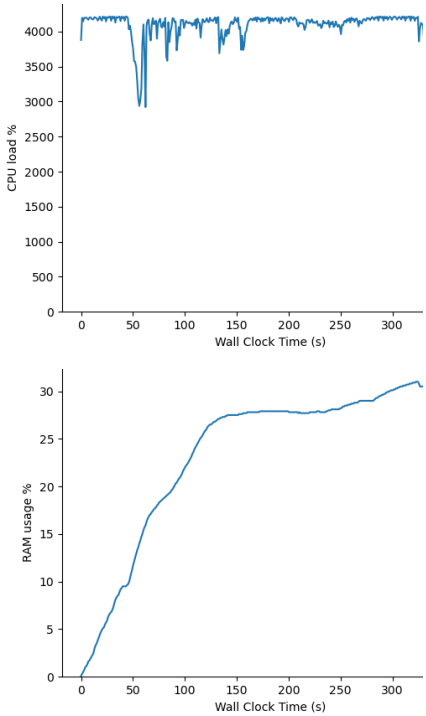
(k) CPU and RAM usage for 33 worker threads



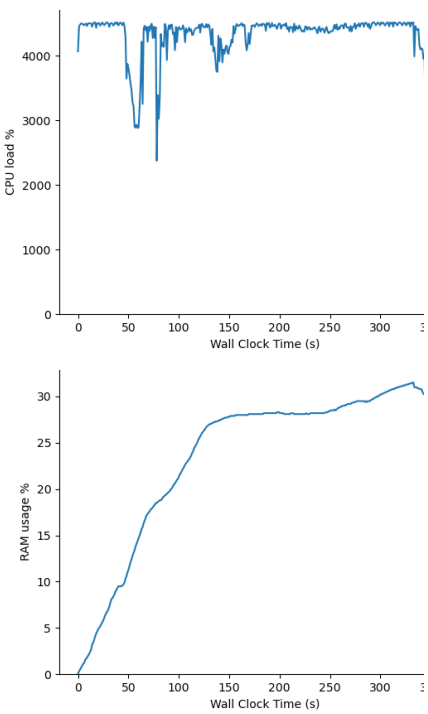
(l) CPU and RAM usage for 36 worker threads



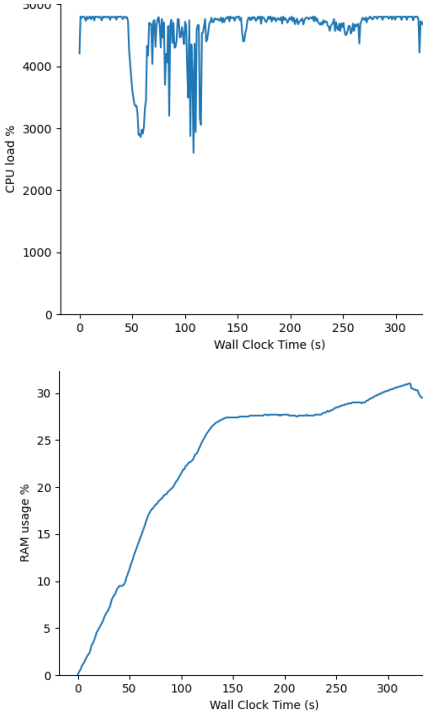
(m) CPU and RAM usage for 39 worker threads



(n) CPU and RAM usage for 42 worker threads



(o) CPU and RAM usage for 45 worker threads



(p) CPU and RAM usage for 48 worker threads

Bibliography

[1] L.F Abbott. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin*, 50(5–6):303–304, Nov 1999.

[2] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, 2015.

[3] Mark W Barnett and Philip M Larkman. The action potential. *Practical Neurology*, 7(3):192–197, 2007.

[4] T. Binzegger. A quantitative map of the circuit of cat primary visual cortex. *Journal of Neuroscience*, 24(39):8441–8453, Sep 2004.

[5] Nicholas T. Carnevale and Michael L. Hines. *The NEURON Book*. Cambridge University Press, USA, 2006.

[6] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.

[7] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[8] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[9] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. July 1978.

[10] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, Dec 1997.

[11] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1912.06680, December 2019.

[12] Alessandro Pellegrini. *Parallelization of Discrete Event Simulation Models*. Studi e Ricerche. Sapienza Università Editrice, November 2015.

[13] Tobias C. Potjans and Markus Diesmann. The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model. *Cerebral Cortex*, 24(3):785–806, 12 2012.

[14] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[15] Renan O. Shimoura, Nilton L. Kamiji, Rodrigo F.O. Pena, Vinicius L. Cordeiro, Cesar C. Ceballos, Romaro Cecilia, and Antonio C. Roque. [re] the cell-type specific cortical microcircuit: Relating structure and activity in a full-scale spiking network model. *Zenodo*, May 2018.

[16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, and et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, Dec 2018.

[18] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017.

[19] Marcel Stimberg, Romain Brette, and Dan FM Goodman. Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314, August 2019.

[20] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Assessing load sharing within optimistic simulation platforms (invited paper). 12 2012.

[21] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Load sharing for optimistic parallel simulations on multi core machines. *ACM SIGMETRICS Performance Evaluation Review*, 40:2–11, 08 2012.