University of Nevada
Reno

Parallel Implementation of a Large Scale Biologically Realistic NeoCortical Neural Network Simulator

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Engineering.

by

E. Courtenay Wilson

Dr. Frederick C. Harris, Jr., Thesis advisor

August 2001

The thesis of E. Courtenay Wilson is approved:

Thesis Advisor

Department Chair

Dean, Graduate School

University of Nevada

Reno

August 2001

Dedication

To Kaija

Acknowledgements

I would like to thank Dr. Frederick C. Harris, Jr. for all of his encouragement and technical guidance in the making of this Master's work. I would also like to thank Dr. Philip H. Goodman for his neuro-biological guidance and James B. Maciokas for his help with channel design. I would also like to thank Dr. Angkul Kongmunvattana for his guidance with parallel computing.

Finally, I would like to thank my family, my friends and my cat for their love and support throughout my Master's thesis work.

Abstract

This thesis discusses the work that was done to create an object-oriented, biologically realistic neocortical-neural network simulator. The object-oriented design for this simulator enables a flexibility in scale and modularity of the system, as well as modeling the relationships between neurons in a given network. The simulator also incorporates laboratory-determined synaptic and membrane parameters into a large-scale simulation, thus modeling realistic cortical modules. Parallel processing of this very large-scale, object-oriented simulator is key for modeling biological accuracy of synaptic and membrane dynamics, while preserving the relationships among neurons. Results show biological accuracy in synaptic and membrane dynamics, as well suggesting that computational models of this scope can produce realistic spike encoding of human speech.

Contents

List of Figures

Chapter 1

Introduction

The primary goal of this thesis work is to create a novel simulator based on biological principles of a mammalian neocortex. Parallel processing of this very large-scale object-oriented neural network is key for modeling biological accuracy of synaptic, membrane, and network dynamics, while preserving relationships among neurons. Clustering algorithms applied to the dense cell connection matrix enable load balancing and data parallelism by organizing groups of cells onto a particular node, thus reducing the performance of inter-nodal communication.

The design of the simulator was done using object-oriented techniques to facilitate modularity, scalability, and portability. This approach allowed us to encapsulate functionality into specific objects and to swap these objects out with others that employed different implementations. We did extensive research into the neuro-biological principles governing cortical cell dynamics in order to accurately represent a large-scale network *in silico*. It was through this research that we understood that the importance of a biological neural network rested in the relationships among neurons in a given community communicating through electro-chemical processes. Our object-oriented design allowed us to model these relationships and to utilize spike encoding in our neural communication paradigm.

Once the simulator was designed and implemented, optimization was performed in several key areas to increase performance and conserve memory. The object-oriented design employed in the development of this simulator allows for a coarse or fine grain parallel model to be run. In order to be biologically realistic, however, a very fine grain parallel model is used to simulate the highly communicative cell/compartment connections within the cortex. It is because of this fine grain approach that a high-speed/low-latency interconnection network is utilized.

The rest of this thesis, is structured as follows: In Chapter 2, we present a review of current technology, including other neural simulators being utilized by researchers, as well as current spike encoding techniques. Chapter 3 gives a background into the biomechanics of neurons, synapses, and channels, as well as the computational design that is based upon this biology. In Chapter 4 we cover our computational results, including a discussion of the parallel implementation. We also show the biological results of the simulator and the biological accuracy of them. Chapter 5 details our conclusions and a sampling of future work to be done on the simulator.

Chapter 2

Review of Current Technology

Currently there are two major tools being utilized by researchers to model neural activity. These two are NEURON[11] and GENESIS[1], and their corresponding parallel implementations. These tools are used primarily to model single cells or small networks of cells in an extremely detailed manner. These tools are excellent for neuroscientists who wish to test or model the behavior of specific compartments within a cell in response to a given input. In Subsection 2.1, we give a more detailed account of NEURON and GENESIS.

In terms of spike encoding, current technology in this area spans a very broad area, ranging from neuromorphic analog VLSI design to theoretical models of memoryless state machines. In Subsection 2.2, we will cover a high level account of these areas.

2.1 Current Biological Neural Network Modeling Tools

The major simulators currently in use today are NEURON and GENESIS. NEURON was developed at Yale University as a collaborative effort between the Computer Science and Neuroscience Departments in the mid-1990's. GENESIS was developed at California Institute of Technology in the early 1990's. These simulations first began with a sequential implementation and the developers have recently added parallel versions. Each of these simulators model the constituent cells in the network in very intricate detail, that is so coupled that the manuals on these simulators say you can only compute small realistic simulations. Both NEURON and GENESIS calculate the Hodgkin-Huxley equations [35](a set of equations used to calculate cell membrane potentials) at each step in the simulation run[1, 11]. Although both simulators are very close in functionality, NEURON provides many line fitting functions, while GENESIS allows the user to add functionality to specific objects by compiling in outside source code.

The cells are modeled in both simulators with precise anatomical accuracy, in that the neuron contains on the order of thousands of dendrites, each with its own highly detailed dynamics being performed[1, 11]. However, in each simulator, synaptic dynamics are virtually ignored. While GENESIS models only a loose Hebbian and anti-Hebbian synaptic weight modification algorithm[29], NEURON does neither synaptic learning nor Redistribution of Synaptic Efficacy[9]. GENESIS does, however, allow the users to write their own script for synaptic modification and incorporate that into the simulation[1].

Because of this fine detail of activity within the single neuron, the overall network of cells - in both NEURON and GENESIS - are very sparsely connected[1, 12]. Thus, the parallel implementations of these simulations utilize a coarse grain parallelism approach, in which one multi-compartment cell is modeled on one processor, Such an example was recently published in [14], where a single Purkenje cell was allocated to a single processor on a Cray T3E. In GENESIS, furthermore, the user must specify in their input file how the cells are to be distributed among the different nodes that are to run in parallel[1]. The parallel implementations of Each simulator utilizes Parallel Virtual Machine (PVM) [6] as it's parallel platform.

In addition, the modeling language in which NEURON is written is a proprietary and interpretive language[12]. To modify the existing NEURON code, or run complex simulations based on an input file, one must learn the language of NEURON first, making scalability more difficult[12].

2.2 Current Spike Encoding Systems

Spike encoding of a particular message is a very important way to send information as it allows a multi-dimensional analysis of data. The encoding of information comes from the combination of several factors including temporal, phase, and inter-spike interval variability[26]. In temporal encoding, the information is encoded in the rate of spikes per some time interval. The reigning hypothesis states that the stronger the signal, or the greater the significance of the signal, the more spikes there will be within a given time interval. This has been shown to be true in regards to the motor cortex, in which a stimulus to a muscle will result in greater spikes in response to greater stimulus [26].

In phase encoding, the placement of the spikes within some interval and their overall oscillatory pattern is used to encode information [26]. In addition, the inter-spike interval is used to encode a third dimension of data. In this case, it is the varying time intervals between successive spikes that encodes data. All three of these combined allows for a three dimensional encoding of data, which is currently in use in satellite communication [5].

In other areas of spike encoding, researchers have developed models to represent learning in a neural network. In [28], the authors present their findings from running a simple simulation using the IQR421 software modeling tool, which is an Artificial Neural Network program. Their primary focus is real-time processing, and as such, small networks are employed. A Fast Fourier Transform (FFT) is performed on the audio input and the excitatory neurons in the network receive additional input from a noise source firing at 10Hz with a Poisson distribution.

In the network, the populations of excitatory neurons project one-to-one onto interneurons. They employed Hebbian learning principles to train the network on audio input. Their results shows that they could process this network in real-time.

Other researchers, such as [4], are creating biologically accurate cortical models within analog VLSI chips. What they are calling a Silicon Cortex (SCX) is a hybrid analog-digital electronic system fabricated with CMOS VLSI technology. They used visual and auditory signals as input because of the analog properties inherent in these waveforms.

Their hardware design was done using an analog VLSI (aVLSI), where the creative algorithm consisted of making as many neurons and synapses in hardware while conserving the costly real estate of the microchip. The neuronal and synaptic design was conducted using resistors to emulate current flow across membranes, and capacitors were used to store state values, thus making hardware the limiting factor.

The communication paradigm had to be optimized for space by handling inter-chip communication with an asynchronous digital multiplexing Address-Event Representation (AER) array. The inter-neuronal communications were tested using one of two systems: hardwired and arbitered. In the hardwired case, the neurons were connected directly to the address encoder, where each spike activates the encoder to send the action potential signal to the destination synapse. This proved to be very high-speed and simple, but collision of incoming spikes was a problem since this produced erroneous destination addresses.

This resource contention of the address encoder lead to the second design, which is the arbitrated communication paradigm. In this system, an arbitrator is interposed between the sending neuron and the address encoder, in which collided spikes are either queued or discarded. This implementation gave the highest throughput but also gave some timing uncertainty because of queuing or discarding of spikes by the arbitor.

The communication itself is handled via an address look up within an object called the Local Address Event Bus (LAEB). The purpose of this is to take the spike event (which is merely a broadcast of the senders address), look up the sender's address in the table, and send this spike notification to the destination synapses corresponding to this sending neuron. Because of hardware limitations, the current maximum number of neurons on any given chip is 10^4.

The communication within the SCX is asynchronous, relying on the random firing pattern of silicon neurons. Because this is implemented in hardware, and most communication is local to the chip, processing is close to real time. However, in a Poisson spike train, timings of spikes will become out of phase, and results of biological accuracy are sacrificed.

In more theoretical models of biological neural networks, [20] presents a new approach to computational neural networks. This theory is a hybrid model of finite state machines (in which neuronal activity is broken into discrete time slices and synchronized accordingly), and integrate-and-fire biological neurons (in which neural activity is asynchronously timed and all neurons store"data" for the current moment only). The authors call their new approach a "Liquid State Machine" (LSM) in which neurons are the liquid and the mathematical principles form the state functions.

The authors stated goal in producing this LSM is to model neural activity in real time and with low memory usage. In order to achieve this, they propose a "memory-less" model where the current state of the liquid represents the present and past inputs to the liquid. Additionally, these liquids are not given multiple time constants for relaxation and all interaction is local to the network.

The authors categorize two main factors important to the model. The first is the separation property (SP), which is similar in design to the inter-spike interval encoding of other systems. The second factor is the approximation category (AP) which governs the transference of internal states to target outputs. It is with both of these factors that the mathematics of the LSM create the memory-less system.

In their results, they show a liquid consisting of 135 neurons, randomly connected with a relative probability, and an input of randomly generated spike trains injected into 30% of the cells. They used perceptron-like learning rules to train the synapses. Their network classified the two Poisson spike trains that were input. Their hope is that this theoretical approach can serve as a basis for novel neuromorphic engineering techniques.

Chapter 3

System Design

In this chapter we will cover the computational design of the neocortical simulator. Section 3.1 consists of a biological background of neocortical mechanics. This biological discussion is necessary for understanding of the design decisions that were made in order to effectively model the cortex. In the computational section, we go into detail of how each object is designed, including what biological functionality is incorporated into certain objects. For a more detailed account of neural processing, please see [25], an excellent text on the subject.

3.1 Biological Topology

The neocortex is organized into structural and functional units, each one containing other elements or performing specific functions. The anatomical constructs are the columns and layers, which contain such physiological constructs as neurons, synapses, and channels. Within the brain, the columns are highly localized areas of neural activity, with each column containing multiple layers. Each layer contains various quantities of heterogeneous clusters of neurons. Each neuron is comprised of multiple dendrites, one soma, one axon, multiple synapses, and multiple channels. The synapses are the connecting points between two distinct neurons and their strength is modulated during neuronal activity. The channels on the other hand are a part of the membrane wall and are active regardless neural communication.

Connectivity among neurons within the brain is best described as being highly connected within layers, less so within a column, and much less connectivity across column boundaries. Neurons communicate with each other through binary events called action potentials, or spikes. No healthy neuron is an island, and it is the relationships between neurons in which all activity occurs. Figure 3.1 shows a set of columns in the visual cortex.

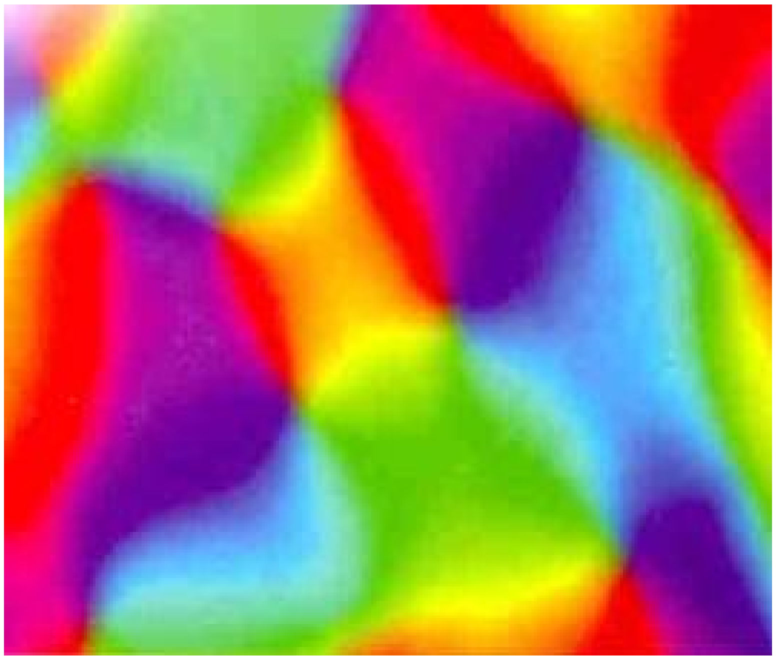

Figure 3.1: A Photograph of columnar activity in visual cortex[25]

3.1.1 Neurons

Physiologically, individual neurons can be grouped into particular categories based upon their membrane properties, channel dynamics, and the types of synapses they employ in connecting to other neurons. For example, an excitatory, Pyramidal cell has a reversal potential of 0 (zero) milliVolts, and connects with other cells using a type of synapse that encourages the receiving cell to become active (excitatory synapse). On the other hand, an interneuron has a reversal potential of -80 milliVolts, and connects with other cells using a type of synapse that dampens the activity of the receiving cell (inhibitory synapse).

A neuron itself is comprised of multiple compartments, such as basal and apical dendrites, a soma, an axon hillock, and an axon. The dendrites form the input structure to the center of the neuron, the soma. The axon hillock is the location on the soma where an action potential, or spike, is generated, and the axon is the compartment that carries the spike out of the neuron to other neurons that are connected to this one. Figure 3.2 shows a photo of a pyramidal neuron.

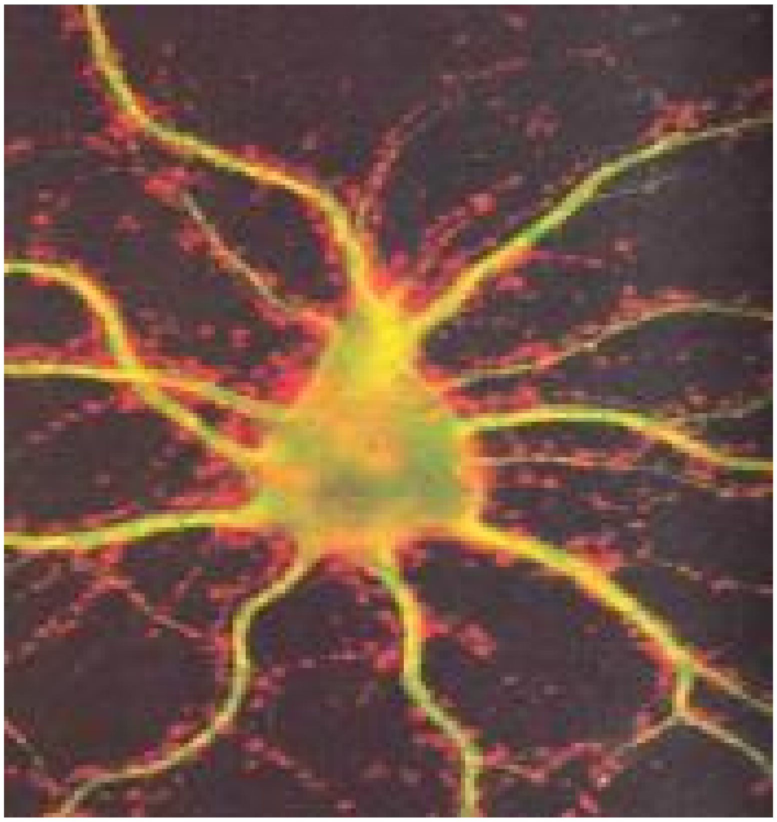

Figure 3.2: A Photograph of a Pyramidal neuron[25]

The channels that each cell utilizes also differ significantly. For example, many interneurons contain channels that suppress cell activity, while the excitatory cells contain only mildly suppressing channels.

Channels line the cell membrane and influence cell behavior, these will be discussed in sub-section 3.1.4. The synapses form the connections between two distinct cells and can attach at any point within a neuron, from the furthermost dendrite to the soma directly, as well as to the axon hillock itself. These will be discussed in Subsection 3.1.2.

In the soma, all incoming currents are aggregated and a decision is calculated as to whether or not threshold has been reached. If it has, then a resulting action potential is to be fired. The action potential is a spike shape of very short duration (on the order of seven to ten microseconds), which ranges in amplitude from the cell threshold (typically -40mV) to 40mV. The spike shape is considered a binary event because it is of such short duration as compared to the waveform output of the synapse, which is on the order to one to two hundred microseconds.

Once a cell fires an action potential, this signal is then output through the axon to all of the other neurons with which this cell connects. This spike shape triggers the synapses on the connecting (post-synaptic) cells to release their neurotransmitters. This chemical response results in a post-synaptic current (PSC) waveform being delivered onto the receiving (post-synaptic) compartment's membrane. The PSC is aggregated into the post-synaptic compartment's total current, and potentially contributes to a new action potential being generated. This begins anew the cycle of a binary event (spiking action potential) triggering an analog event (the PSC), which triggers a binary event, and so on. This process of aggregation and action potential generation is called "integrate-and-fire"[15] and it is through these steps that cells within a given network communicate with one another. For those wishing to learn more about this process of integrate-and-fire, please see [15] and/or [16].

3.1.2 Channels

The channels on the membrane of the cell contribute to propagating the Post Synaptic Current (PSC) from the synapses by opening/closing themselves to allow/block the entrance of certain ions. This ion flow changes the membrane potential of the compartment, and input to the soma, contributing or inhibiting the generation of an action potential. All channels to be discussed here are potassium (K^+) channels and most contain activating (m) and inactivating (h) particles [13, 35]. Figure 3.3 shows a photograph of a channel.

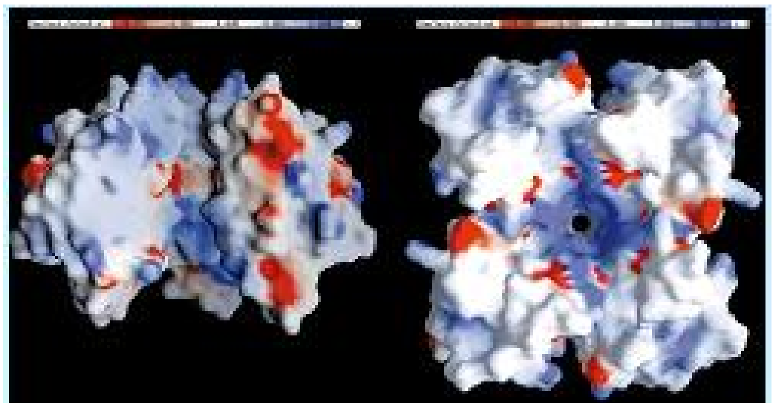

Figure 3.3: A Photograph of a channel, left shows side view, right shows top-down view [18]

The M-Channel, of the K^+ channel family, has only an activation particle. This channel becomes active when an incoming spike shape is received by the parent cell. Its general behavior is that of inhibiting its parent cell from reaching threshold, resulting in a suppression of the output spike shape. Since this channel does not contain an inactivation particle (h), it reaches a steady state of activity gradually, relative to the activation time constant. This can be seen in [9] where the output spike train becomes a regular pattern (constant inter-spike interval).

The A-Channel, also of the K^+ channel family, helps the cell to wash out background noise. It does this by opening more as the cell approaches threshold, and then shutting off once threshold has been reached. Unlike the M-Channel, this one contains an both and activation (m) and an inactivation (h) particle, which allows its dynamics to turn off after a certain time period. This activity can be seen in [9] where there resulting spike train of the parent cell has shows delayed behavior.

The AHP-Channel (After Hyper Polarization Channel) is a voltage independent, Calcium (Ca^{2+}) dependent channel. Like the other channels of the K^+ family, this channel confers a generally suppressing behavior onto its host cell. The name of this channel is derived from the functionality of the channel. The hyper-polaraizing process occurs when the membrane voltage makes a drastic jump from a very high value (+40mV) to a value that is between the resting potentials of the host cell and the channel. This is the negative dip in the spike shape waveform.

With each outgoing action potential, Ca^{2+} enters the cell, forcing the channel to open more, which hyper-polarizes the cell's membrane voltage. This channel slows the spiking rate of its host cell down as the resulting action potentials progress, until the cell reaches a steady state of spiking. The effect of the AHP channel can be seen in [9] where the channel accommodates the cell spiking (gradually increasing inter-spike intervals until a steady state is reached).

3.1.3 Synapses

The synapses are the contextual filters whose effectiveness is modified based on the timing of the input to each cell. They are also responsible for converting the binary action potential input signal into a resulting analog post-synaptic current

(PSC) via neurotransmitter release. The synapses are classified as either excitatory or inhibitory. The excitatory PSC is a positive waveform which contributes positively to the post-synaptic cell reaching threshold and firing an action potential. The inhibitory PSC is a negative waveform, and suppresses the post-synaptic cell's ability to reach threshold, thereby inhibiting a resulting spike.

The effectiveness of each synapse is based on the amount of input it receives from the pre-synaptic cell, and on the amount of signals outgoing from the post-synaptic cell. There are two major algorithms that determine the amount of neurotransmitters released at each pre-synaptic signal, one is a short term dynamic in which the synaptic efficacy is modified within a specific time window, and the long term dynamic of Hebbian Learning. The amount of neurotransmitters ready to be released is quantified by a percent availability term called Utilization of Synaptic Efficacy, or USE (hereafter called `U_mean`). When a synapse releases more neurotransmitters, its signal is amplified; and conversely, when a synapse releases less neurotransmitters, its signal is dampened.

The short term dynamics govern how much neurotransmitters a synapse releases within a given time window. The overall quantity of neurotransmitters released within this window remains the same, however, the distribution of synaptic efficacy is decreased or increased depending on the initial value of `U_mean` and modulated by the depression and facilitation time constants. For example, in a train of incoming spike shapes, the synapse will release less neurotransmitters in response to each spike if the following conditions hold: The depression time constant (tau) is greater than the facilitation tau, and the `U_mean` is large enough to accommodate significant decay. Conversely, the amount of neurotransmitters released with each spike will be increased if the facilitation tau is higher than the depression tau, and the baseline `U_mean` value is low enough to have room to grow. This is considered a short term dynamic because the overall effectiveness of the synapse is not modified permanently, as can be seen from the graphs which describe this dynamics. The area underneath the curves of each facilitation and depression are the same, but the amount of neurotransmitters is merely redistributed.

The Hebbian learning algorithm is another set of principles that determine the effectiveness of a given synapse [32]. Unlike the short term dynamics, this algorithm is dependent on both pre- and post-synaptic spike timing. This algorithm is described as follows: There is a given window in which positive and/or negative learning can take place, and at the center of this moving window is the resulting post-synaptic spike of the cell. The position of the pre-synaptic spike within this learning window determines the updating of the `U_mean` value.

If the pre-synaptic spike occurs within the positive learning window (before the post-synaptic spike), then the `U_mean` value is increased, thus strengthening the connection between these two cells. Conversely, if the pre-synaptic spike occurs within the negative learning window (after the post-synaptic spike), the `U_mean` value is decreased, thus weakening the connection between these two cells. Additionally, if the pre- and post-synaptic spike occur at the same time, or the post-synaptic spike occurs outside either the positive or negative window, the `U_mean` value remains unchanged.

Through Hebbian learning the synapse will strengthen a connection in which the pre-synaptic cell directly contributed to a post-synaptic spike, and will weaken a connection in which the pre-synaptic cell made a negative contribution to the post-synaptic action potential.

3.2 Computational Topology

The computational topology is closely modeled after the biological principles mentioned in Section 3.1. In this simulator the user specifies a biological and/or administrative template for each segment of the model and the program creates the amount of elements needed, as well as creating their inter-relationship [3].

For example, the brain template specifies the duration and frequency of the simulation (administrative) and which columns to include and their inter-column connectivity (biological). Other objects, such as the spike shape and post synaptic conductance (PSG) waveform are modeled through static templates. The choice for making some processes into templates was done to enhance performance on a very large-scale networks, thus the trade off between network size and computational complexity.

The choice for an object-oriented design for this simulator was made because the biological brain is segmented into distinct but interrelated parts. The object-oriented paradigm allows the simulator to model objects generically, changing their behavior through the input parameters without affecting the underlying object functionality. Operation and reporting is based on parameters specified in a text input file. In this way, a user can rapidly model multiple brain regions merely by changing input parameters.

The object-oriented design also allows the user to scale the network size more easily. By changing only the input file, this simulator can model very large numbers of cells and various connections strengths which affects the amount of synapse objects and communication. The design also allows for a modeling of very large numbers of channels and external stimulus all within a given network.

This system design enables object modularity, in which one object implementation can be swapped out for another because functionality is encapsulated within that object. For example, we have employed different communication paradigms by swapping out the MessageBus object with another MessageBus object that implements communication differently.

On an implementation level, there are several basic designs for the objects within the simulator. Some are containers for other objects and maintainers of administrative data; others are both containers and state machines; and others are simply state machines, and do not contain any other objects. Secondarily, there are other objects that perform specific functions important to the computational side, but not directly affecting the biological system. Of the objects described here, the state machines are very important to the simulation as they perform very specific biological tasks that directly impact the behavior of the system and from which biological data is garnered.

In the following subsections, we will describe the object-oriented design of the simulator. Each primary functional object will be illustrated along with a discussion of its relevance to the system as a whole, and its relationship to other objects within the model.

3.2.1 Brain Object

The brain object is both a container of other objects and a repository for administrative data. It is responsible for ensuring that each owned object is visited and updated. It contains the column, layer, cell, stimulus, report, and the synapse helper objects.

At each time step in the simulation the brain visits the following objects: Each stimulus object (to send data to the cells); visits each report object (to collect data on the cells); visits the MessageBus object (to ensure communication between the cells on any node within the cluster); and visits each cell/compartment object (to process the input and change the state of the overall simulator).

On the administrative side, the brain is responsible for such duties as calculating the computational duration of the simulator (in ticks per second), and keeping track of the passage of simulation time. The brain object also holds the location of the signal trap, allowing the user to exit gracefully from the program at any point in time, without losing output report data. Figure 3.4 shows an object-oriented layout of the brain object.

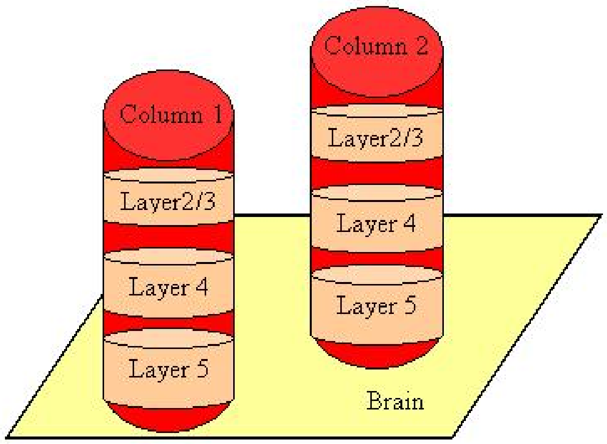

Figure 3.4: A Representation of the Brain.

3.2.2 Cell Object

The cell is another container object; it holds all of the compartment objects for the particular neuron being modeled. The cell object is responsible for ensuring that the messages delivered to it are passed on to the appropriate compartment, whether that message originated outside the cell or within the cell (from a sibling compartment). The cell is also responsible for ensuring that all of its constituent compartments are visited, to allow compartmental processing of their data. Of these contained compartments, only the soma is the minimum required to mimic a point to point model. Figure 3.5 shows a representation of a neuron.

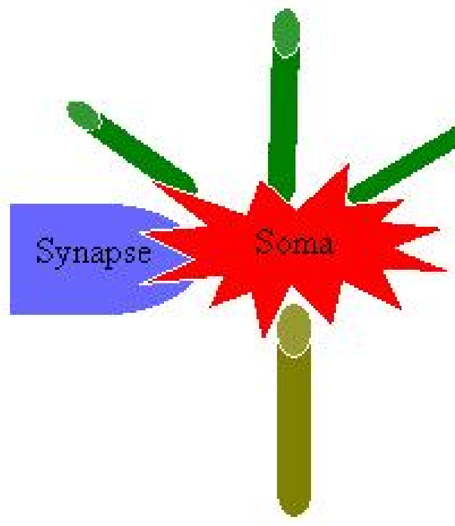

Figure 3.5: A Representation of a Neuron.

3.2.3 Compartment Object

The compartment object is both a container and a state machine. The compartment object contains the synapse, channel, and spike shape objects. It is a state machine in that its membrane voltage is calculated and updated at each time step.

It is a generic object whose functionality is based upon user specified input parameters. For example, only one compartment object is used to model all of the compartments within the cell - specifically, the basal dendrites, the apical dendrites, an axon and a soma. Everything is based on the variables that were specified in the user input file. This includes whether or not this particular compartment will fire an action potential, and if so, which spike shape it will utilize, and to which cells it will send this binary spike event.

The compartment object houses functionality that makes up the heart of the cortical simulator. All the integrate-and-fire routines exist here, all action potentials originate here, and all synapses and channels are updated here. The compartment changes state at each timestep regardless of external input, because it's membrane voltage degrades by a pre-set amount towards resting potential.

3.2.4 Synapse Object

The synapse object is owned by the compartment object and is functionally a state machine. Its state change is triggered by an input from its host cell. This input can take the form of either a pre-synaptic (incoming) spike shape, or a post-synaptic (outgoing) spike shape.

This object incorporates all of the short term dynamic (facilitation and depression) equations [9], as well as all of the Hebbian learning algorithms [29]. The PSG waveform is modeled as a template that is input by the user. It's amplitude is modulated by a scaling factor, and is typically on the order of 100 microseconds in length. The decision to model the PSG waveform as a template was made in order to cut down on computationally expensive calculations that would consume process time in a very large network. In this way certain biomechanics were traded for performance and scalability.

3.2.5 Channel Object

The channel object is also owned by the compartment object and is also primarily a state machine. The channel is a generic object that is designed to model any type of channel. It's state change takes place at every time step in the simulation, unlike the synapse object, whose state change is triggered by a specific event.

The channel equations in use are modeled after channels discussed in [35], which utilize the activation and inactivation particles, voltage and tau values to calculate channel current. This current is output to the host cell at each timestep in the simulation. The channels currently being modeled include the following: Non-Inactivating Muscarinic Potassium Current (M Channel); Delayed, Rectifying Potassium Current (A Channel); Non-Inactivating Calcium-Dependent Potassium Current (AHP Channel).

Each channel, when visited by the compartment, is given the compartment's membrane voltage at that timestep, and in the case of the AHP channel, the Calcium level of the compartment. In the process of the channel current calculations, certain parameters are saved for use in the next state.

3.2.6 Stimulus Object

The stimulus object is of the category of objects that perform specific functional tasks but are not necessarily state machines. This object is used to mimic signals that originate from external sources, such as a voltage or current clamp, as well as ongoing activity from neurons in other areas of the brain (not currently being modeled explicitly).The stimulus object is able to inject into its recipient cells either a voltage or a current that is calculated at pre-determined time interval.

The stimulus object is designed to model the following types: a file based audio input (normalized from decibels to frequency, and then converted into stochastic current stimuli)[24]; a linear (ramp) function; a pulsed (staircase or constant) function; and a sinusoid waveform function. Superposition of multiple stimuli allows the user to mimic background noise in a community of cells, or to enhance synaptic learning.

The destination of these stimuli is determined by another object called the StimulusInject, whose lifetime is limited to the initialization phase of the simulation. The StimulusInject allows for a probabilistic determination of receiving cells. This design allows us to input the same stimulus into multiple groups of cells without having to create another stimulus object, thus conserving memory and enhancing performance.

3.2.7 Report Object

The report object is another object that performs a functional task but does not change state. It handles the data processing of what to report on, which cell/compartment to receive this report, at what frequency (in time steps, not seconds), and at what points in simulation time the report is to be generated. Currently, the user is able to report on a range of information about the compartment, including: voltage; stimulus current; synaptic current; channel current; leak current; net current; and current from adjacent compartments. The parameters of what to report on, which cells to receive this report request and what output file name are all specified in the user input file.

The report object communicates with the cells via the MessageBus object. The report requests are generated in the report object when it is visited by the brain object during the main simulation loop and sent to the MessageBus. The MessageBus in turn delivers the report request to the specified cells/compartments. Finally, the reported data is handed off to the FileIO object for processing in a separate thread.

3.2.8 FileIO and PostProcessing Objects

The FileIO object handles the file input/output within a separate thread from the computation thread. It outputs the data from a cell/compartment to a given temporary file (located on the local hard drive of the node in the cluster). This temporary file is later processed by the PostProcessing object, which formats it into a more user friendly display. The PostProcessing object deletes the temporary file from the local hard drive, and a final, user friendly output data file is sent to the head node for viewing by the user.

3.2.9 Message Object

The Message object is a collection of data that is used in communication between all objects within the simulator. All of the necessary messages are instantiated as blank/empty messages at program inception, and pointers to this pool are passed around by various objects. The implementation of this will be discussed in Chapter 4.

3.2.10 MessageBus Object

One of the most important functional objects is the MessageBus object [2]. Its detailed model will be discussed in Chapter 4, however, it is responsible for all of the message passing between cells, stimulus and cells, reports and cells, and cells and FileIO. It encapsulates all of the Message Passing Interface (MPI) [8, 23, 30] calls, and is responsible for all inter-nodal communication. It is utilized at all stages of the simulator, from initialization to simulation end. It is also responsible for synchronizing the globally distributed set of brains, so as to prevent race conditions, deadlock and starvation among the nodes on the cluster.

Chapter 4

Results

In this Chapter, we present some of the results that have been gathered from designing and implementing an object oriented neocortical simulator. The first half of this chapter discusses the parallel implementation, which includes a subsection discussing the algorithm for distributing the data onto multiple nodes within the cluster. The next subsection discusses message passing and communication, and the MessageBus object in particular. The following subsection discusses the hardware of the system, with a justification for the need of such large amounts of RAM and such high-speed interconnection network. Then, in Subsection 4.1.3, optimization issues are discussed and the results of scaling the system. The final subsection of the first half of this chapter details the control flow of the system with a listing of the activity of each object that is involved at each stage.

In the second half of this chapter, the biological accuracy of the simulator is demonstrated. In these subsections, synaptic dynamics of short term `U_mean` modification and Hebbian learning are illustrated. There is also a display of the membrane dynamics of various channels that have been modeled. Finally, in Subsection 4.2.3, there is a display of the results of the auditory processing of a spoken word. These results show raw decibel input, the spike encoded input, and the density graph of these spikes.

4.1 Parallel Implementation

Because of the potential size of the neural network, this simulator was designed to run in parallel. The parallel algorithm spreads data across multiple nodes and allows for fine or coarse grain communication between these nodes. Distribution of the data takes place in initialization and consists of aggregating specific groups of cells onto a given node. Specifically, the program divides the number of layers as evenly as possible across all nodes. Connection information is maintained by the individual compartments only.

Inter-cell communication is handled via the MessageBus object, which functions as a router for packing and delivering of the messages. It encapsulates all of the Message Passing Interface (MPI) calls, and can easily be swapped out with a MessageBus object employing an alternate communication paradigm.

4.1.1 Message Passing and Communication

The MessageBus object is an important object for facilitating communication between objects within each brain. Through the use of MPI, the current implementation handles communication with external nodes. The communication scheme of the simulator is described by Figure 4.1.

When a compartment fires an action potential, it communicates this event by creating a message for each compartment that is to receive this notification. The compartment then passes this message either to it's parent cell (for intra-cell communication), or to the MessageBus (for inter-cell communication). In the former case, the parent cell delivers the message to the sibling compartment without going through

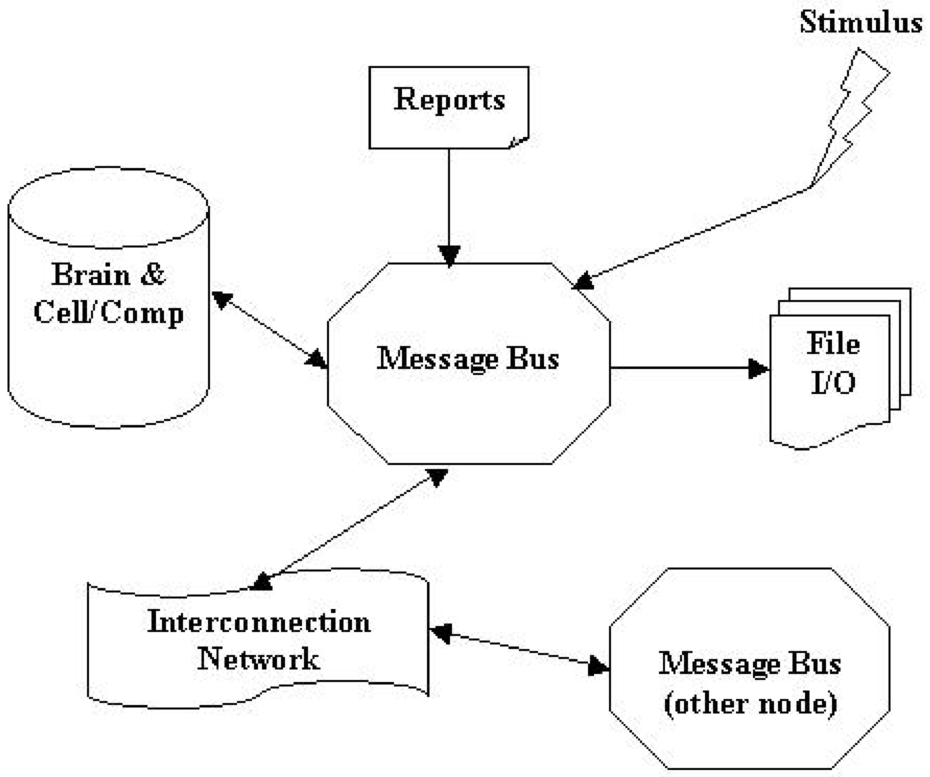

Figure 4.1: Communication scheme of the simulator

the MessageBus. In the latter case, however, the MessageBus takes the message and relays it to the cell specified in the message address.

If the receiving cell is off-node, it is the responsibility of the MessageBus to package this information in an MPI data buffer and send it to the proper node. Upon receipt of an off-node message, the MessageBus unpacks the data from the MPI data buffer, stores it in a Message object, and then forwards it to the local, recipient cell.

The message that is sent between cells contains two things (1) a static set of data that is consistent for each message, and (2) a dynamic set of data that changes depending on the message type. The static data set contains the message type, sending address, receiving address and timestep (this makes up the message envelope). The dynamic data set contains such information as the membrane voltage values, external stimuli values, or reporting information.

The sending and receiving of messages is done using blocking sends (`MPI_Send`) and blocking receives (`MPI_Recv`) to allow for deterministic communication. The MessageBus employs an algorithm for sending and receiving off-node messages that prevents node starvation or deadlock. This algorithm is based on a simple all-to-all personalized message exchange as described in [19]. Each node sends a message at each time step; the MPI element TAG is utilized to distinguish between data messages, empty messages, and multi-part messages. The MessageBus exits this stage only when the sign off has been given by each node in the cluster. In this way, the MessageBus prevents a situation whereby one node in the cluster has not finished sending its messages but the other MessageBuses have finished receiving.

The MessageBus also uses synchronization to create a lock step barrier, so that the brains on different nodes reach the MessageBus at the same time, to process messages in the same order. This barrier prevents a deadlock condition (one MessageBus is sending but no other MessageBus is receiving), as well as preventing a starvation situation (one MessageBus is receiving, but no other nodes are sending).

4.1.2 Hardware and Interconnection Network

In our model, connectivity between cells drives everything from memory and CPU usage to latency in inter-nodal communication. The cluster that runs this simulation, `cortex`, consist of 30 dual-Pentium III 1-GHz processor nodes, with 4GB of RAM per node. This hardware architecture permits high connectivity (more than 10 million synapses per node). In addition, Myrinet [22] is utilized to handle the intensity of communication that occurs in the fine grain parallel model.

Without such a high-bandwidth/low-latency interconnection network, the system's communication would saturate at low inter-column connectivity. For example, this simulator was run on another cluster, `hercules`, which consisted of 20 dual Pentium II 750 MHz processors, each consisting of 512MB of RAM, and a dual Fast Ethernet interconnect[5]. On this cluster, the simulator was able to run with low connectivity (low numbers of synapses and messages), but went to swap once the number of synapses approached one million per node. Also on this cluster, when running a fine grain distributed model, the machine did not go to swap, but CPU utilization of the processes fluctuated between "running" and "sleeping." This was tracked down to reveal the flooding of the fast Ethernet interconnect network.

4.1.3 Optimization

Thus far there have been two major areas of optimization. These did not change the interface to any other object, but did change the internal implementation of the object being optimized. The first change dealt with our use of messages. Because there is a direct correlation between connectivity and message passing, we needed to rethink the message allocation and de-allocation within the system.

When we first implemented the simulator, the messages were created and deleted using the standard object-oriented “new” and “delete” as the simulation progressed. Profiling of the simulator showed that this was a very time consuming task and so the following optimization was made. A MessageManager object was created as a warehouse of message data. During initialization of the program, the MessageManager allocates all of the messages that will be utilized throughout the entire run. The amount to be allocated is dependent on 1) the number of synapses, 2) the number of cells receiving stimulus, and 3) the number of cells receiving reports. Since all three of these objects utilize the MessageBus for message passing, a maximum of all three quantities denotes the maximum number of messages that could be passed at each time step. In case there is a need for more messages, however, the MessageManager can allocate more messages if they are needed at any point in the simulation. Because multiple threads access this shared message pool, the MessageManager utilizes a system semaphore to block access to the pool when this re-allocation is occurring.

In terms of functionality, the MessageManager wraps the “new” and “delete” function calls, and merely returns a pointer to the next available message in the queue. If no more memory can be allocated, the MessageManager returns a NULL value. This optimization resulted in a 30% increase in performance time.

Another important optimization was the breaking down of the Synapse object into static and dynamic parts. Initially, the synapse object was instantiated with both parts, but profiling showed that each synapse was approaching 1500 bytes in size. This was a huge amount of memory being allocated, considering that a modest simulation has approximately 1.5 million synapses, therefor utilizing 2.25 GB of RAM just for synapses. In order to run larger synapse counts, and hence larger network sizes, we needed to optimize this object.

In reviewing the data required for the synapses, we noticed that there was only a small amount that was absolutely necessary within the synapse itself. This small amount was kept in the local synapse object as a dynamic data set. The rest of the data was deemed to be unchanging and pulled out into the brain object where only one copy was instantiated. The brain object now provides wrapper functions for this data set. This reduction of the synapse object to 120 bytes, which is more than an order of magnitude smaller, allows the simulator to accommodate larger quantities of synapses, and therefore, larger and more highly connected cellular networks.

4.1.4 Scaling and Performance Analysis

In order to test scaling of network size, several test cases were run both sequentially and then in parallel. In each test case, connectivity was the driving factor. For the sequential test cases, a single column was created based upon biological data gathered from [21]. The connectivity ranged from a “full column,” which constituted approximately 1.5 million synapses to a “mini-column,” which contained approximately 800,000 synapses; to a “micro-column,” which contained approximately 34,000 synapses. These test cases were run on both cortex (the 30 node 4GB RAM/node cluster) and on hercules (the 20 node 512 MB RAM/node cluster).

Figure 4.2 shows these results plotted against each other. As one can see there is a faster execution time on cortex which is associated with its faster processor. This is to be expected, however, when the synapse count exceeds 1 million per node, hercules must go to swap to accommodate the required memory allocations, and this causes its performance time to slow down tremendously. This simulation was stopped after

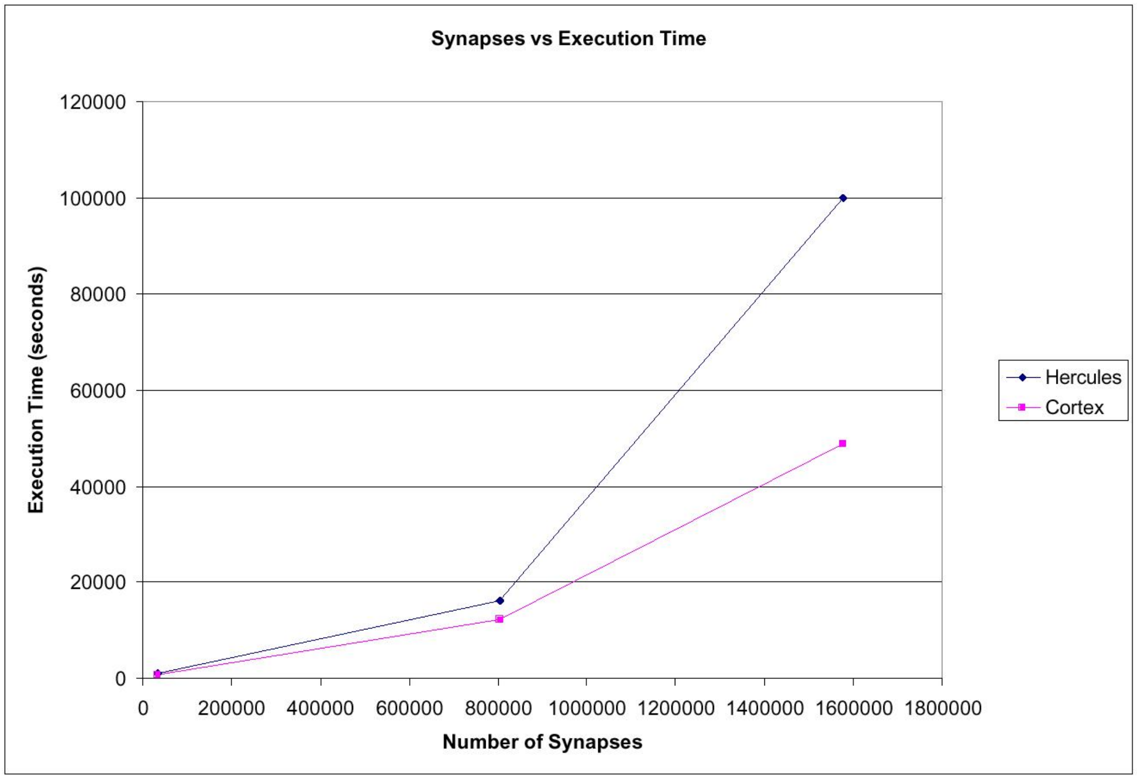

Figure 4.2: Execution time versus synapses on Hercules and Cortex using Ethernet, the execution time for 1.5 million synapses on Hercules is a fake number, since the program never finished

it had reached only timestep 100 (out of 5,000) after approximately 3 days of running.

To test the inter-connection network, several test cases were run on both the Ethernet (100Mbps) and the Myrinet interconnect. Connectivity was modulated in each case. A five column run, fifteen column run, and a twenty-five column run were performed using two different cell connection schemes, where each column was partitioned onto its own node. There were intra-column connections based upon biological realism[21] and the inter-column connections were from excitatory neurons in LayerB of one column, to excitatory neurons in LayerB of another column.

In the first scheme, the multiple columns were daisy-chained together, with each column receiving external stimulus. In the second scheme, the multiple columns were connected with each other in an n-squared topology. This means that each column was connected to every other column with the same probability, and that there were intra-column connections as well. There were approximately 5,100 cells per column, connected with a constant probability, resulting in approximately 65,000 synapses per node. Figure 4.4 shows the results of these simulations for each node quantity using Ethernet interconnect. Figure 4.3 shows the results of the n-squared connectivity with the Myrinet interconnect.

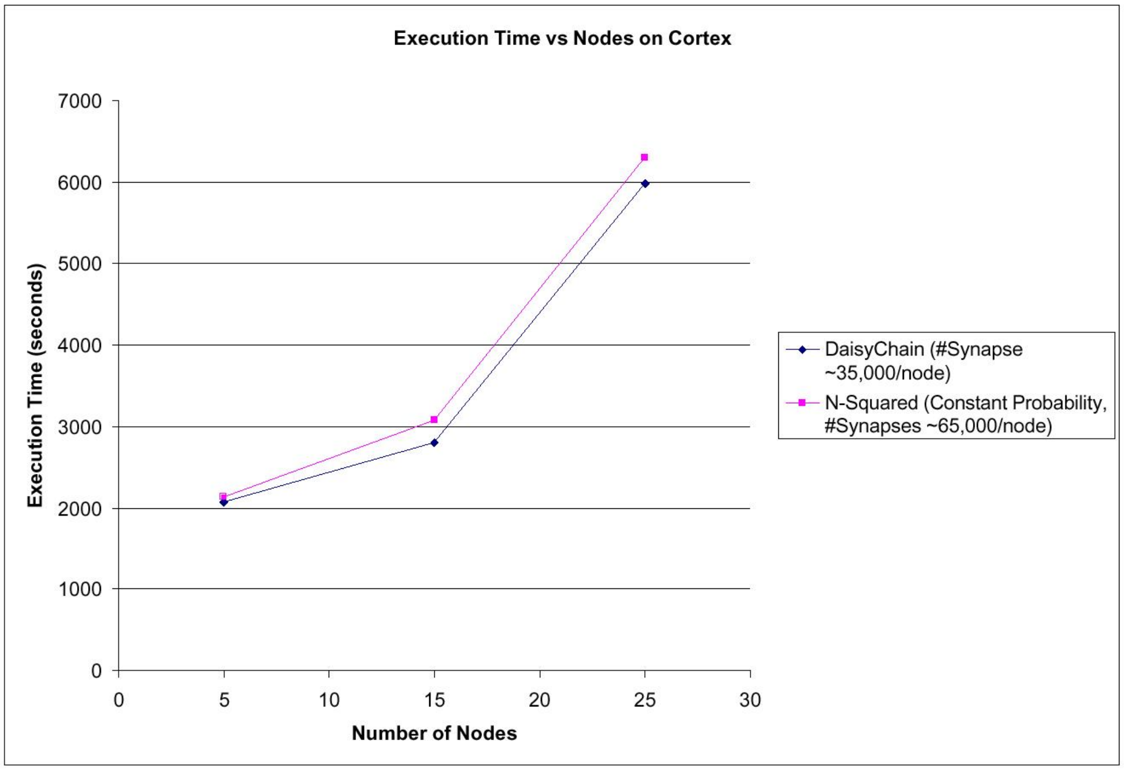

Figure 4.3: Scaling results on Cortex using Ethernet with various connection schemes.

4.1.5 Control Flow

The control flow of the cortical simulator consists of three major stages: initialization, brain computation, and post-processing of the data. Each stage and its

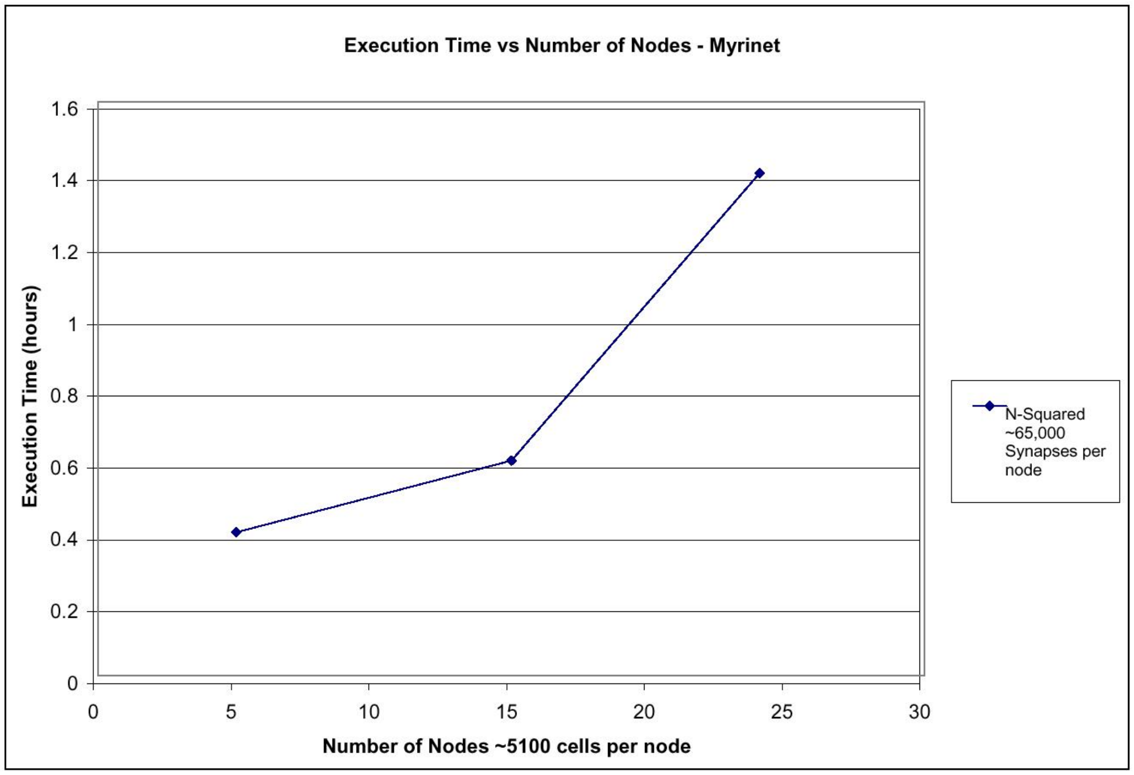

Figure 4.4: Scaling results on Cortex using Myrinet with various connection schemes.

constituent parts are discussed in some detail within this section.

The initialization stage is handled by the InitializationManager and is implemented as a factory model. Initialization occurs on all of the nodes concurrently. In designing the system, we decided to duplicate work across processes in order to save idle time on non-root nodes, as well as to avoid communicating the CellConnectionMatrix once it had been instantiated. Each step in initialization is contingent on successful completion of the previous steps, and the program may exit at any point in this process if an error is detected.

The first step of initialization is the reading and parsing of the input file. This file contains templates for biological data structures (i.e., cells, channels, synapses, columns, layers, brain, etc.), functional units (such as reports, stimulus), and connectivity. The data from the input file is stored in temporary variables which utilize the dynamic memory allocation of Standard Template Library (STL) vectors.

The second step of initialization is the displaying of values to the user. The DisplayObject object, if requested by the user, will prompt the user to accept/reject the configuration that has just been read in.

The third step of initialization is the error checking of the temporary variables. The ErrorCheck object is responsible for examining the data for consistency and biological accuracy. In case of errors in the input, this object will display the incorrect variable along with the correct range for that variable, before exiting the program.

The fourth step in initialization lays the groundwork for parallelization of the simulation. This step is handled by the CreateConnectMatrix object, which is responsible for filling in the global connectivity matrix. CreateConnectMatrix utilizes the connection schemes that were specified by the user and stores this information in the CellManager object. The CellManager holds the groups of cell types, their location in the brain (node number, column and layer), and their global cell identification values. The CreateConnectMatrix object updates each cell list in the CellManager with the constituent cell connections.

The fifth step creates the DistributionManager object, which uses the connection information now stored in the CellManager to distribute the various cell groups onto the nodes within the cluster. During this process the DistributionManager updates the CellManager with the node number for each cell group. The CellManager can then be used by the MakeObjects object to instantiate the cell objects.

The sixth step in initialization is handled by the MakeObjects object, which instantiate each object from the biological template provided by the user. Each node in the cluster has a MakeObjects object which is responsible for checking the current machine number, and making only those cells which are situated on that particular node. The CellManager object is utilized heavily for this task since it stores the node address of each cell being instantiated. This step creates the data parallelization of the cortical simulator. The global cell list is now a virtual global list, in which its constituent parts are spread out among different nodes within the cluster.

The last step in initialization is handled by the ConnectionManager object and consists of making the connections between the cell/compartments in the brain. Each node loops through it's local cell list and fills in the connection matrix for the given cell/compartment. One synapse object is instantiated for each connection, and it is owned by the receiving (post-synaptic) cell/compartment. The MessageBus is employed at this point to allow the compartments to communicate and process the synaptic ID values of their connecting cell/compartment.

After the initialization stage is complete, each constituent helper object is deleted. The brain objects on each node are then synchronized before beginning the next stage of the simulation and the file I/O thread is launched. The simulator is now ready to proceed to the main computational loop.

On each timestep, the brain visits the objects that it contains. These include the stimulus, report, MessageBus (which is actually a global object, but is pointed to by the brain), and each local cell. The cells then visit their constituent compartments, which, in turn, process the integrate-and-fire routines mentioned previously.

The third stage of the simulator is the PostProcessing of the temporary data files. It takes place either once the brain object has completed its pre-determined number of cycles or once a report has completed its duration and the temporary file has been closed. This stage consists of using the PostProcessing object to reformat the output data files into a more user friendly context. Once the final stage has been completed, the simulation deletes all allocated memory and the program terminates.

4.2 Biological Accuracy

4.2.1 Synaptic Dynamics

Short Term Synaptic Dynamics In order to test the short term dynamics of `U_mean` modification algorithm, a stimulus of a preset pattern was injected into a single cell. This stimulus consisted of six pulses of amplitude 2.1 nanoAmps (nA) sustained for 0.002 seconds each, followed by a 0.5 second pause, and then a final single pulse of the same amplitude and width as before. This pulse injects enough sustained current into the recipient cell to induce an action potential. This cell was then connected to on other cell and the membrane voltage of the post-synaptic cell was measured to determine if the PSC waveform had been affected by the input stimuli.

The next four figures show the results of these runs with various synaptic, short-term dynamics turned on. Figure 4.5 shows a baseline activity with no dynamics turned on. As one can see, the membrane voltage remains peaked at a constant value, reflecting that the PSC was not modified in any way.

Figure 4.6 shows the cell voltage when the synapse has facilitation time constants higher that the depression time constants, and that the `U_mean` has room to grow. With each incoming spike, the synapse releases more neurotransmitters (it is facilitated to do so), and after a pause, the synapse has recovered its quantity of neurotransmitters.

Figure 4.7 shows the cell voltage when the synapse has depression time constants higher than facilitating ones, and when the `U_mean` has a higher initial value, and has room to degrade. With each incoming spike, the synapse releases less and less

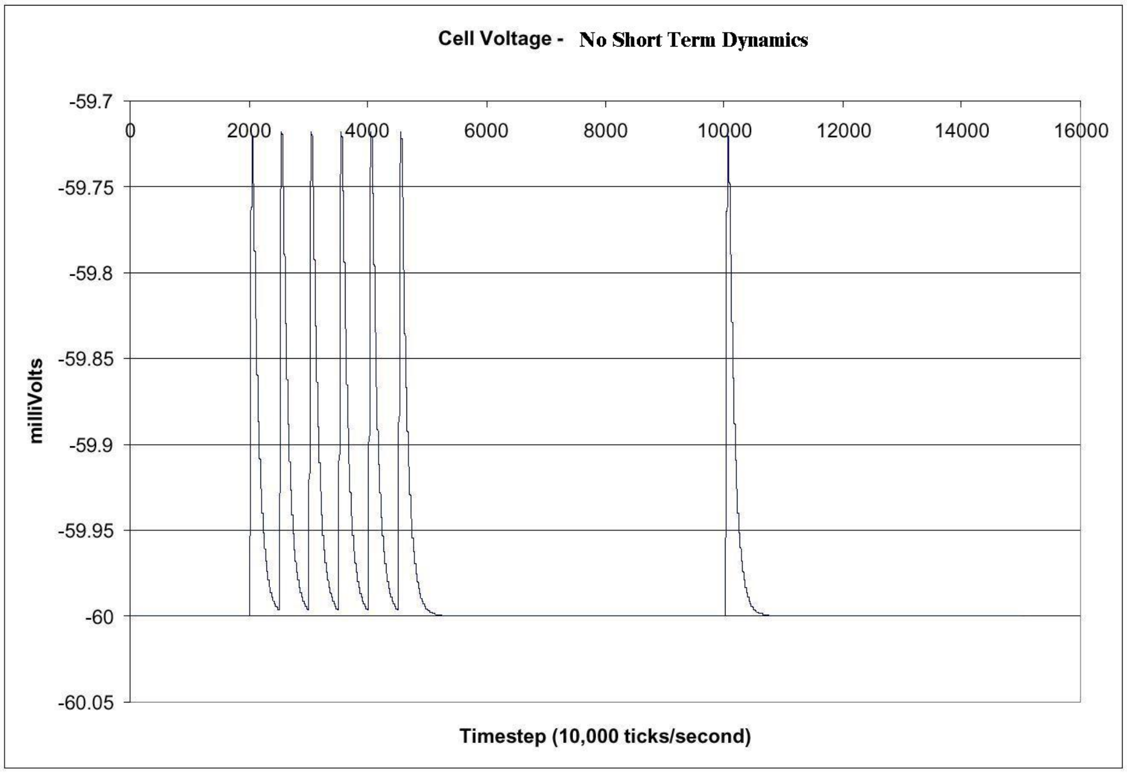

Figure 4.5: Cell Voltage with no short term dynamics enabled.

neurotransmitters (its effectiveness if depressed), and after a pause, the synapse has recovered some quantity of neurotransmitters. This the synaptic PSC waveform has a lower amplitude each time, affecting the post-synaptic cell less and less with each spike in a train. We can see that the amplitudes of the cell voltage waveforms decrease over the six pulses.

Figure 4.8 shows the cell voltage when both facilitation and depression dynamics are active. This shows the neurotransmitters degrading at first and then reaching a steady state value. After a pause, however, the synapse has recovered its neurotransmitters.

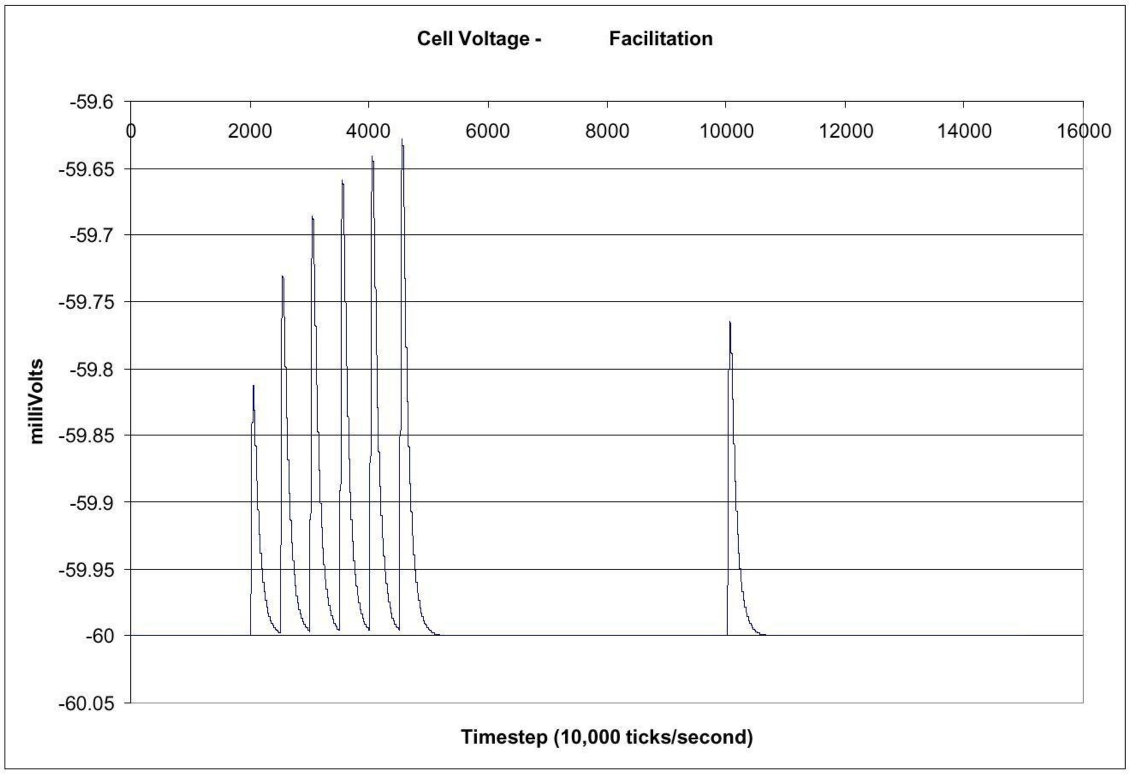

Figure 4.6: Cell Voltage with short term Facilitation enabled.

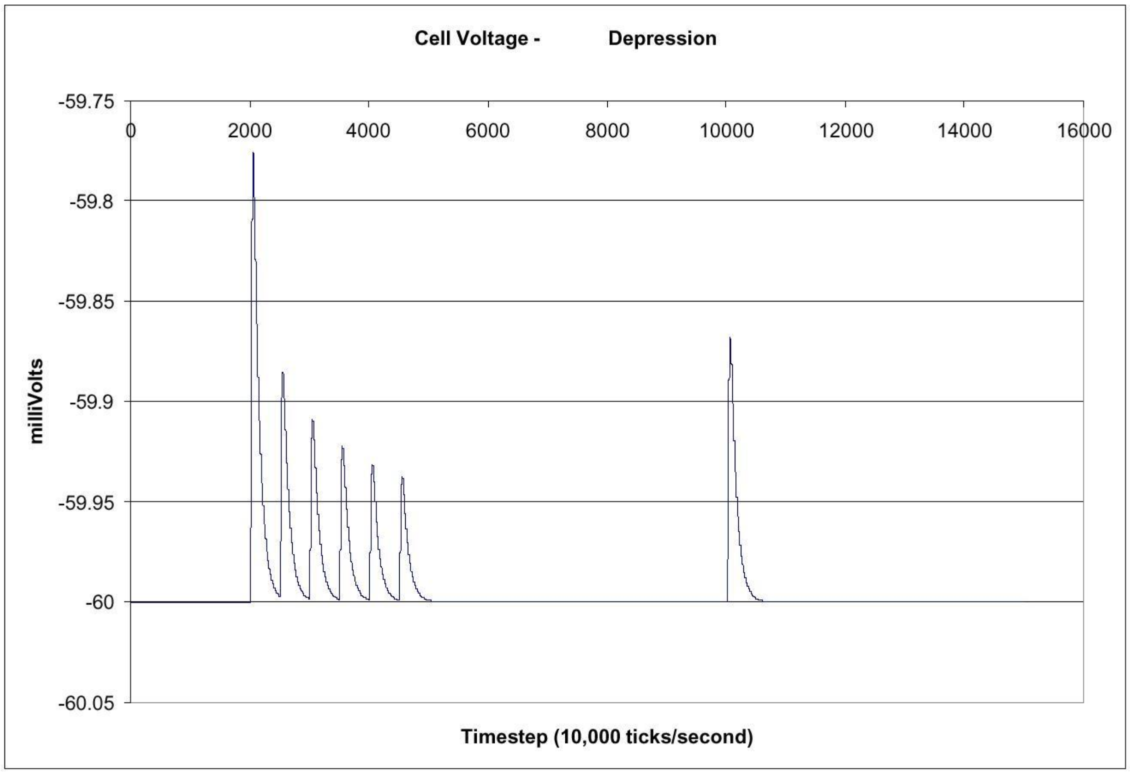

Figure 4.7: Cell Voltage with short term Depression enabled.

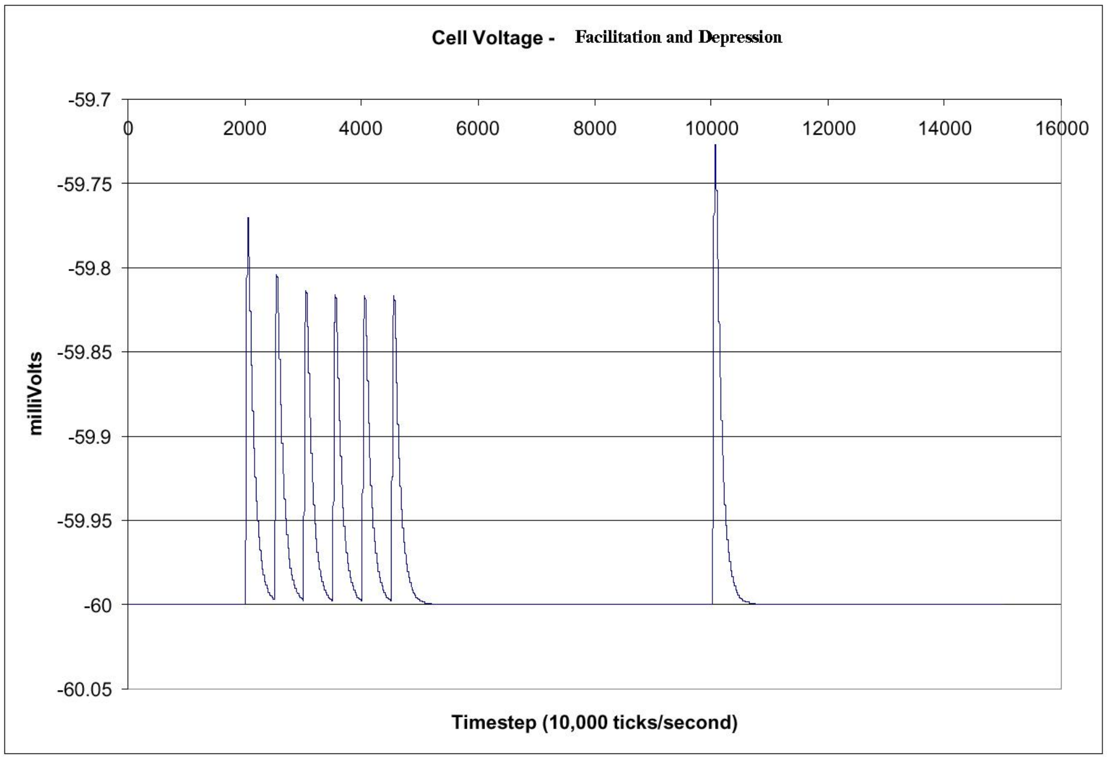

Figure 4.8: Cell Voltage with short term Facilitation and Depression enabled.

Hebbian Learning in Synapses To test learning, a similar network topology was employed using two cells connected point to point. In this case, however, both the pre- and post-synaptic cells are receiving an external stimulus. This stimulus is a pulse set at 2.1 nA with a sustained width of 0.002 and delivered as six pulses, a 0.5 second pause, and then a final pulse. These stimuli are sent to the pre-synaptic cell, and then the same pulse waveform is sent to the post-synaptic cell, but this stimulus is shifted in time.

The direction of this shift determines how the learning is tested. For example, to test positive Hebbian learning, we want to positively reinforce the connection between the two cells. This means forcing the post-synaptic cell to fire an action potential within the positive Hebbian window occurring after the pre-synaptic cell has fired a spike. We shifted the inject time of the post-synaptic input stimulus to coincide with the positive Hebbian learning window. This causes the post-synaptic cell to strengthen the connection (increasing the `U_mean`) with the pre-synaptic cell.

Conversely, to test negative Hebbian learning, the pre-synaptic cell is given the stimulus that causes it to spike and therefore send a signal to the post-synaptic cell. This post-synaptic cell receives a stimulus within the negative Hebbian learning window (which occurs before the spike of the pre-synaptic cell). This causes the post-synaptic cell to weaken the connection with the pre0synaptic cell (decreasing the `U_mean`).

In Figure 4.9, we show a baseline of cell voltage with no learning dynamics turned on. As one can see, the cell spikes at regular intervals (corresponding to its stimulus injection), and that its membrane voltage has minor peaks around those spikes. These minor peaks correspond to the input from the pre-synaptic cell, and since there is no modification of the synaptic `U_mean`, the peak of each of these remains constant.

In Figure 4.10, we show the results of positive Hebbian learning. The post-synaptic cell fires after receiving a pre-synaptic input, and the connection is strengthened by increasing the `U_mean`. This can be seen in the increase in amplitude of the cell voltage (minor) waveforms.

In Figure 4.11, we show the results of negative Hebbian learning. The post-synaptic cell fires before it receives pre-synaptic input, and this connection is weakened by decreasing the `U_mean` value. This can be seen in the gradual decrease of the non-spike cell voltage waveform amplitudes.

Figure 4.12 shows the results of both positive and negative Hebbian learning. the post-synaptic cell fires within a window that sometimes corresponds to the negative Hebbian window, which weakens the connection between the two cells. At other

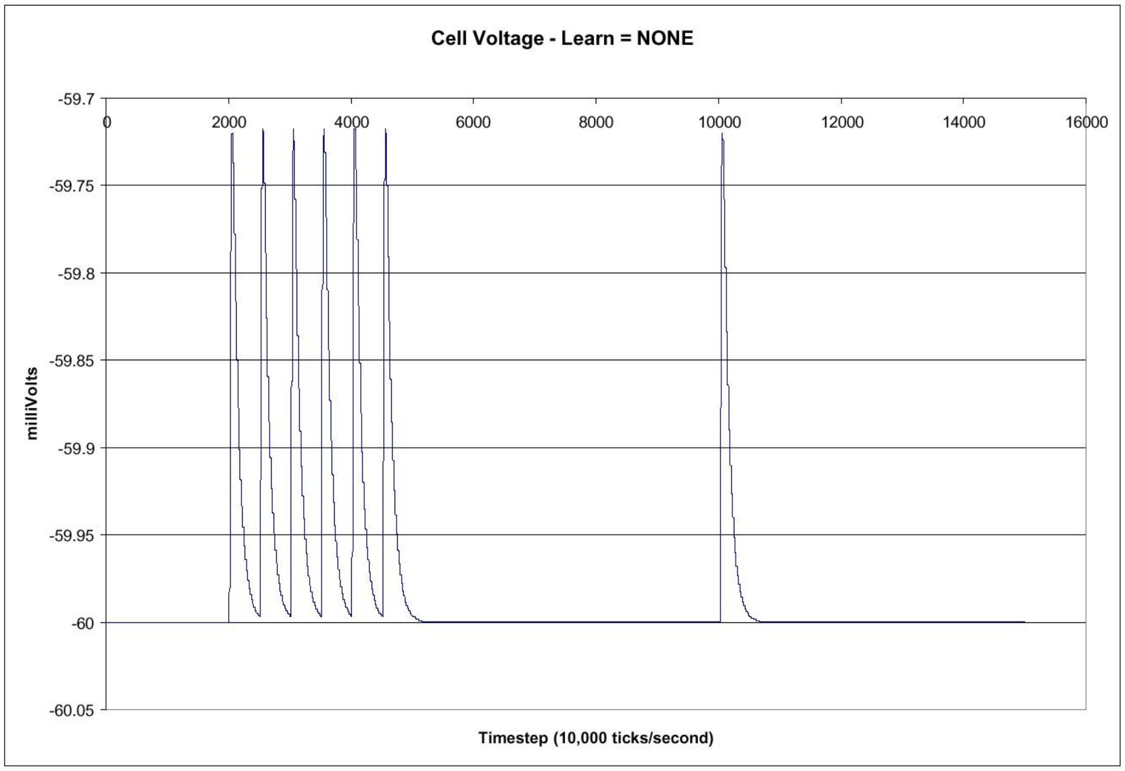

Figure 4.9: Cell Voltage with no learning enabled.

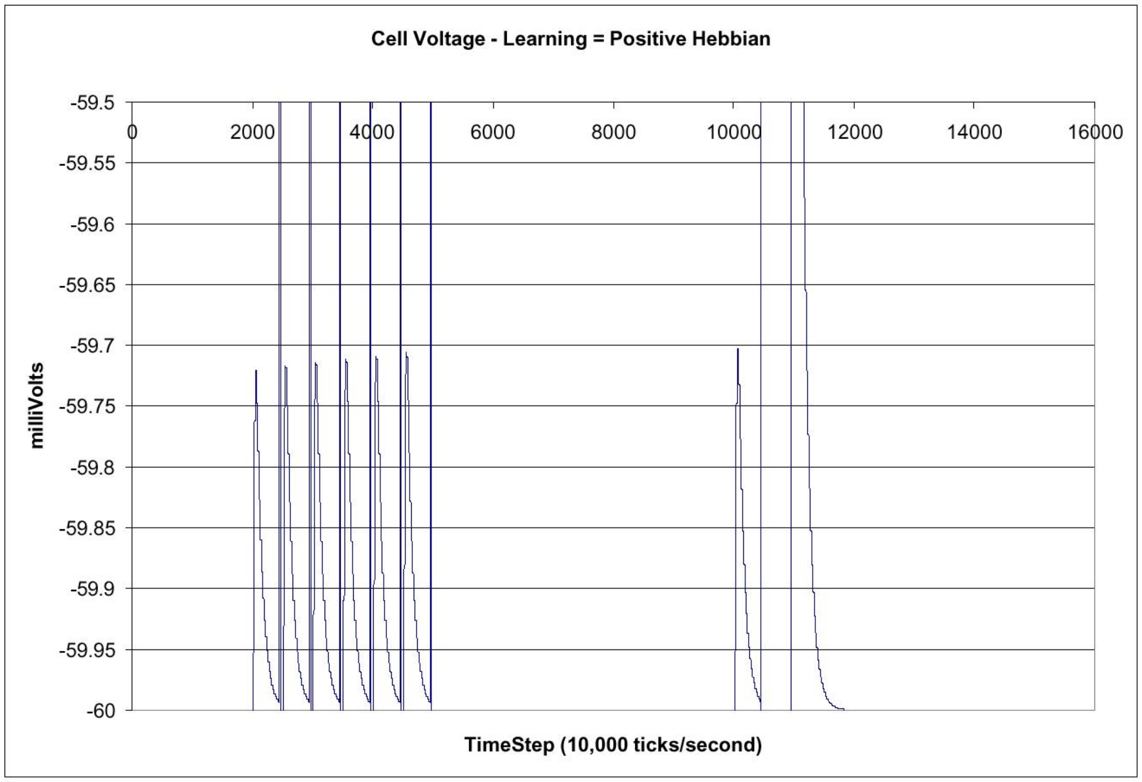

Figure 4.10: Cell Voltage with positive Hebbian learning enabled.

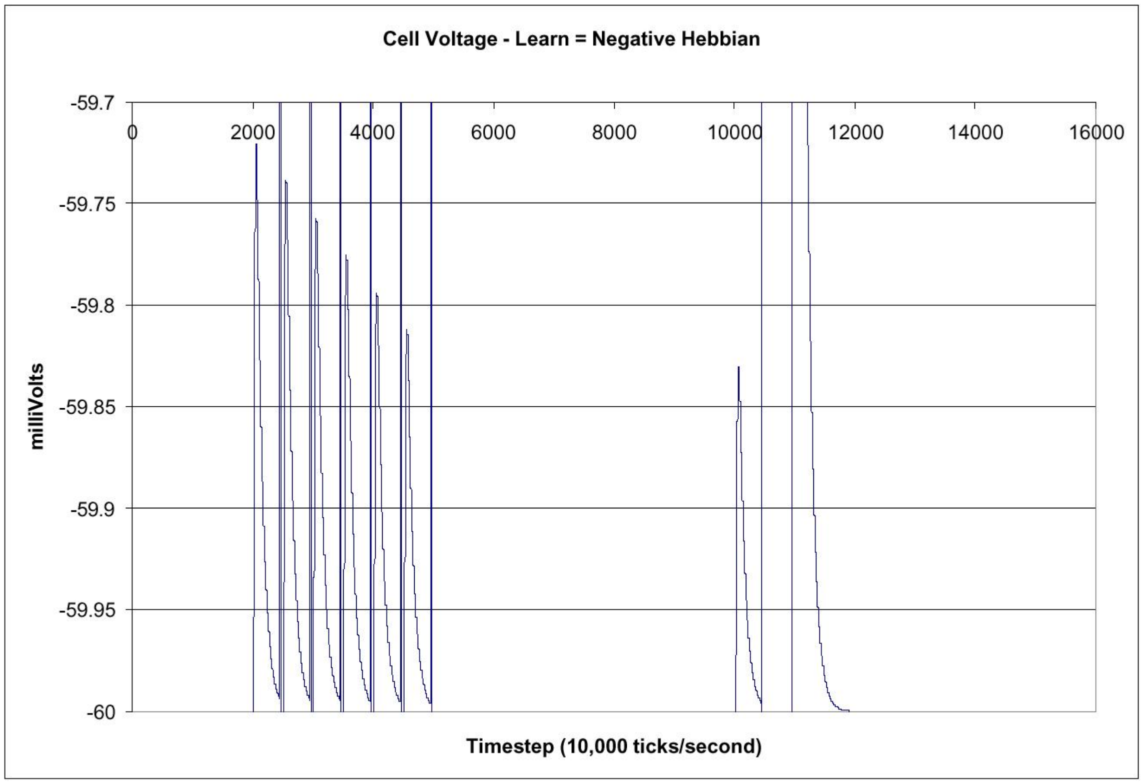

Figure 4.11: Cell Voltage with negative Hebbian learning enabled.

times, the post-synaptic cell fires within a window that corresponds to the positive Hebbian learning window, which strengthens the connection. This can be seen in the graph where the amplitude of the cell voltage waveforms (non-spike) oscillate between higher and lower peak values.

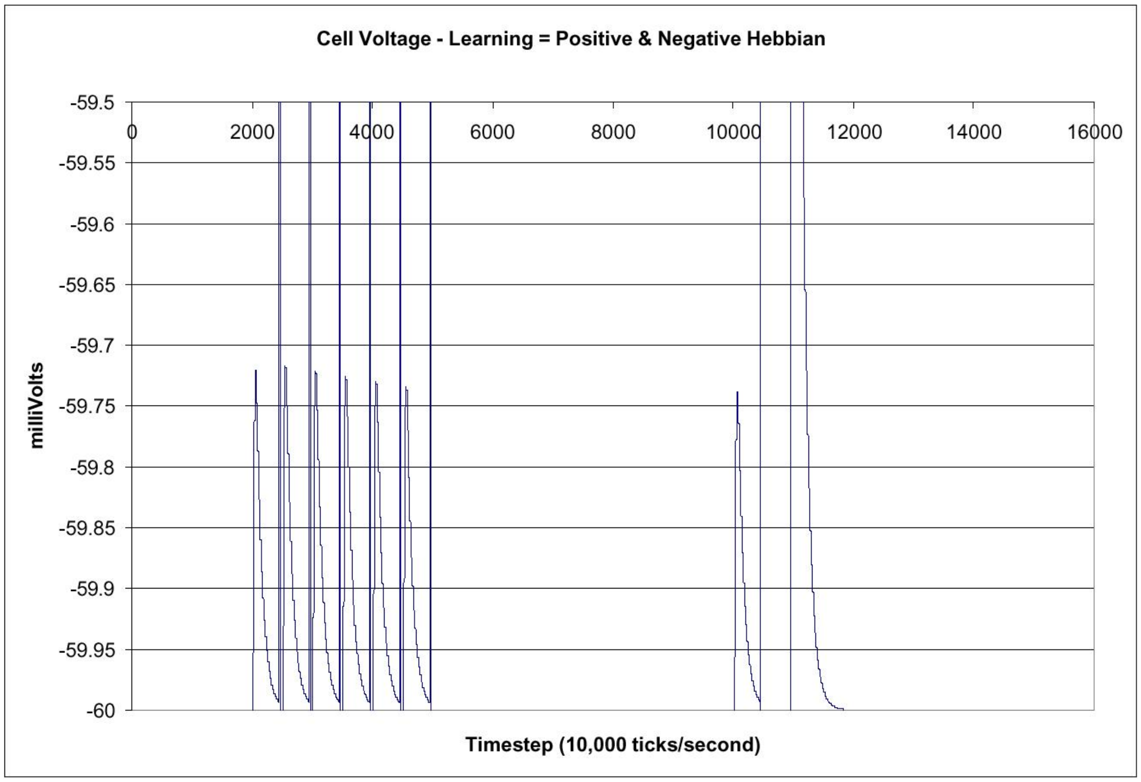

Figure 4.12: Cell Voltage with positive and negative Hebbian learning enabled.

4.2.2 Channel Dynamics

To test the channel dynamics, a series of pulses were input into a single cell model. This cell contained the channel to be tested, and the input pattern of an 800microseconds pulse width of 0.03 nA, and then a 200 microsecond gap of no input stimulus followed by a second 800 microsecond pulse of 0.07nA. The tau values, the half min values, the slope factors, and the strengths were modified to get the results necessary to comply with the biological accuracy presented in [9].

In Figure 4.13, we show the cell activity when there are no channels in the compartment. As one can see in the second half of the simulation (after 15,000 timesteps), the cell has an average spiking rate of 20Hz. This figure is a baseline for comparison of the activity of the cell when certain channels are being modeled.

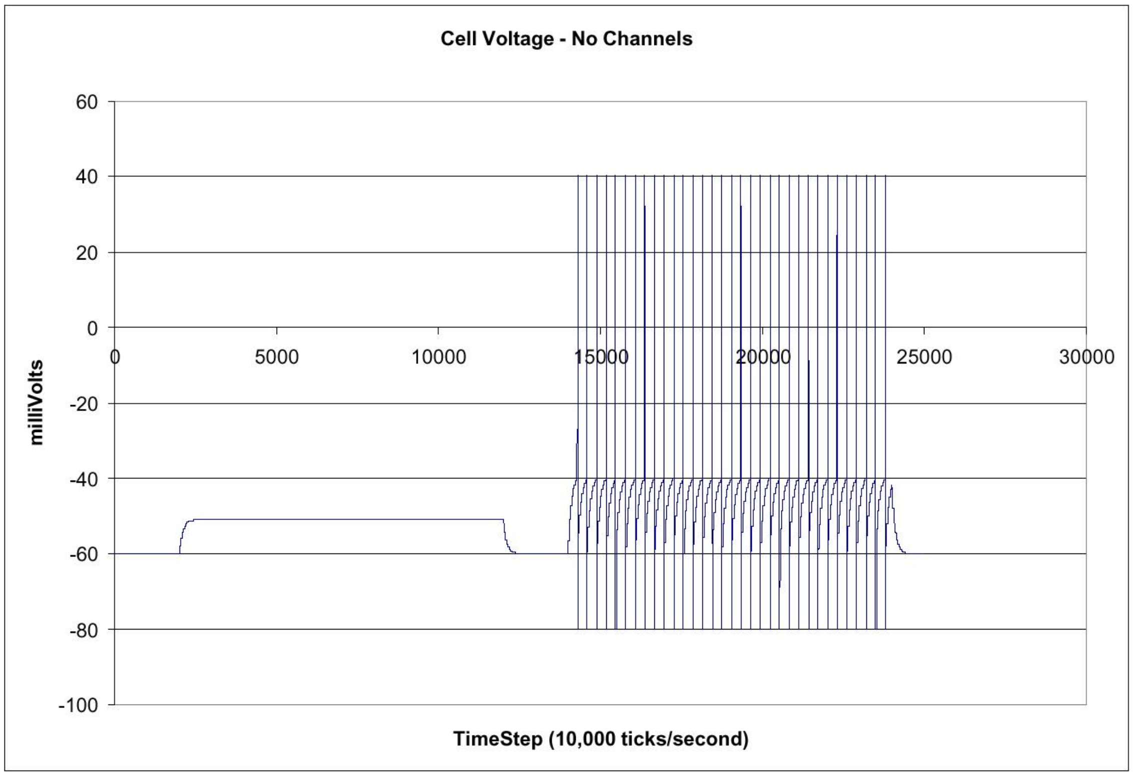

Figure 4.13: Cell Voltage with no channels.[33]

In Figure 4.14, we show the cell activity when a bursting channel (bAC) is employed in the cell. This is done by modeling a Sodium (Na^+) channel in addition to an AHP channel. The Na^+ channel is modeled as a modified A Channel where the reversal potential is changed from -80mV to 100mV. It can be seen from this figure that because of the Na^+ channel, there is bursting behavior, but when the Na^+ channel shuts off, the AHP channel then accommodates the cell behavior.

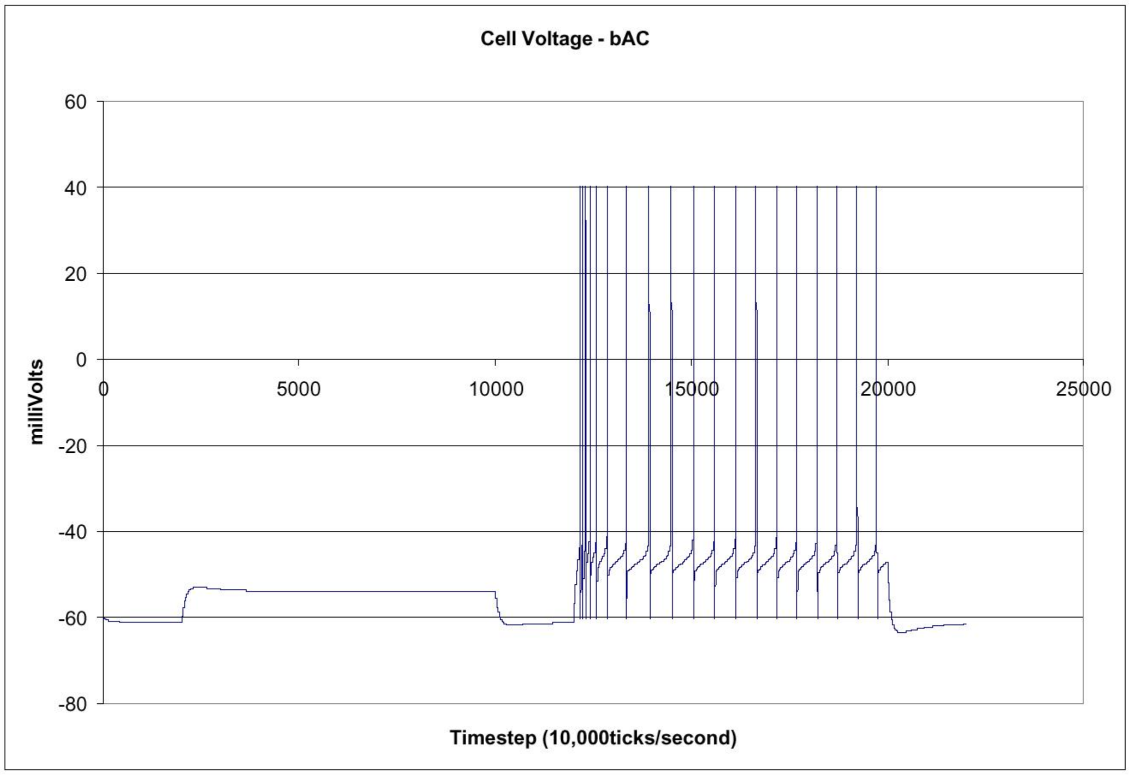

Figure 4.14: Cell Voltage with bursting accommodating channel (bAC).[33]

In Figure 4.15, we show the cell activity when we have a classic accommodating behavior (cAC). In this case, we modeled this by using an AHP channel, and one can see from the figure that there is a gradual suppression of cell spiking resulting in a steady state being reached around timestep 17,000.

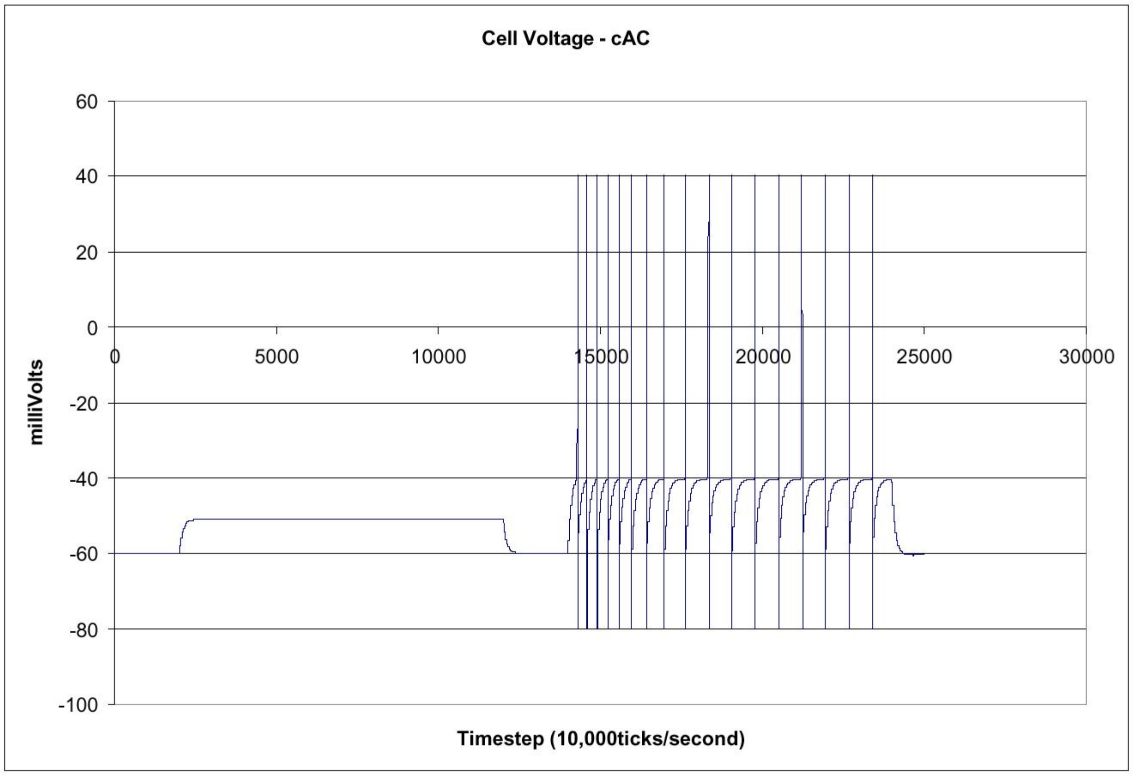

Figure 4.15: Cell Voltage with classic accommodating channel (cAC). [33]

In Figure 4.16, we show a classic non-accommodating (cNAC) behavior using an M channel to model it. As can be seen, the cell reaches steady state very quickly, which is what it means to be non-accommodating, since accommodating behavior takes longer to reach steady state.

In Figure 4.17 we show the cell activity with a delayed non-accommodating (dNAC) channel in use. This was modeled using an A channel, which activates sub-threshold with a fast time constant for the m-particle (suppressing cell firing).

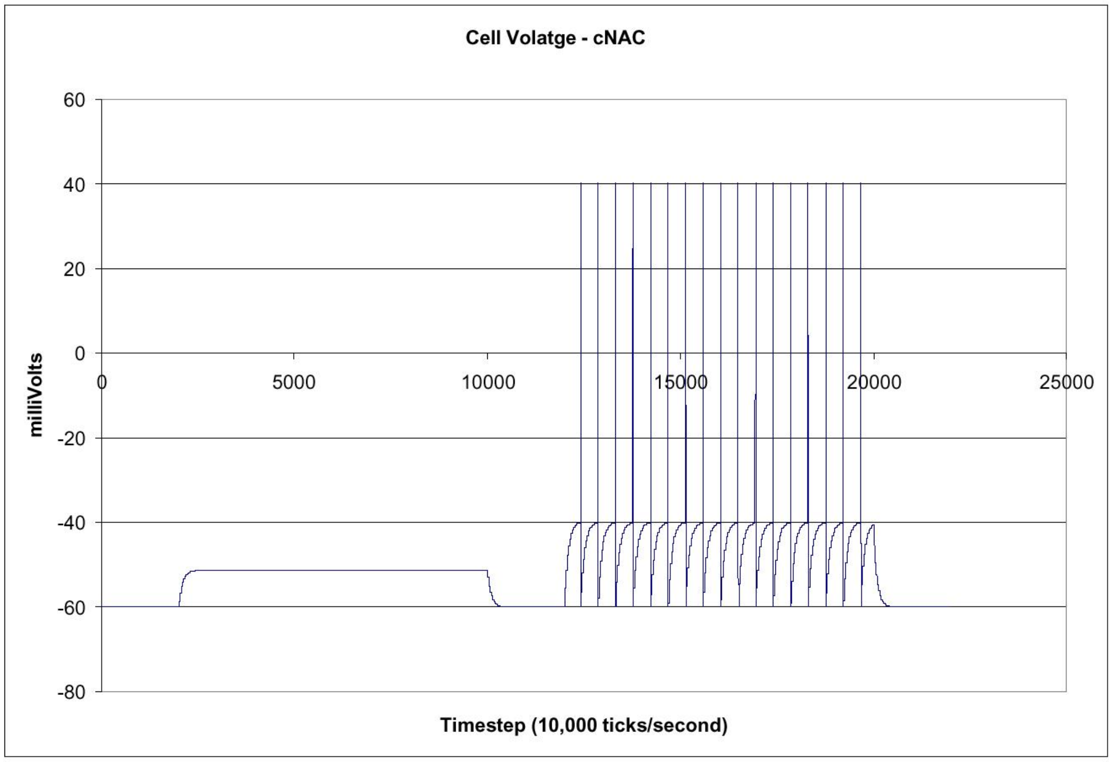

Figure 4.16: Cell Voltage with classic non-accommodating channel (cNAC) [33].

Since the h particle has a slower time constant, it gradually shuts down this channel, which allows it to spike. So the name comes from the delay (d) in spiking and the non-accommodating behavior can be seen in the constant inter-spike interval.

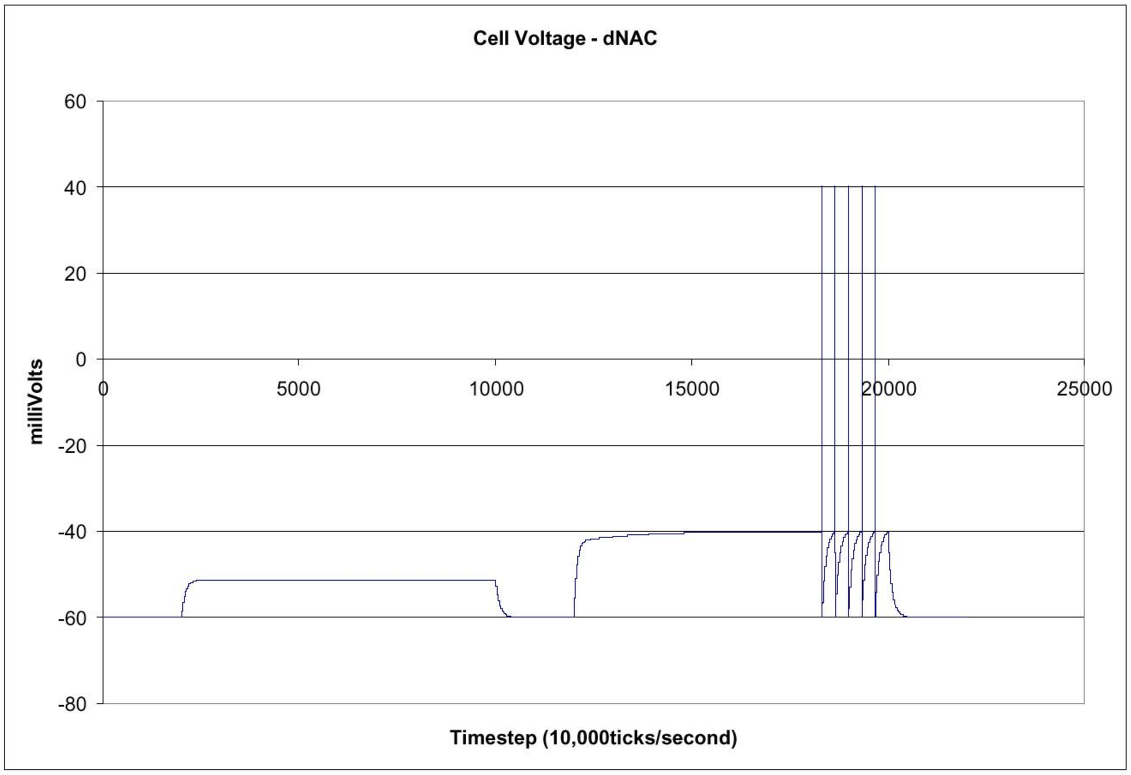

Figure 4.17: Cell Voltage with delayed non-accommodating channel (dNAC) [33].

Figure 4.18 shows the cell behavior when a classic stuttering (cSTUT) channel is employed. This is modeled using a SIK channel, which is a Supra-threshold Inactivating Potassium (K^+) channel (using the A channel dynamics). In this case, the m particle does not open until the cell spikes (as can be seen in the figure, from time steps [14,000..15,000]), which causes the channel top open, and thus to suppress cell spiking activity (as can be seen in the figure, from timesteps [15,000..18,000]). Then the h particle becomes effective in closing the channel, which allows spiking, and this process repeats itself.

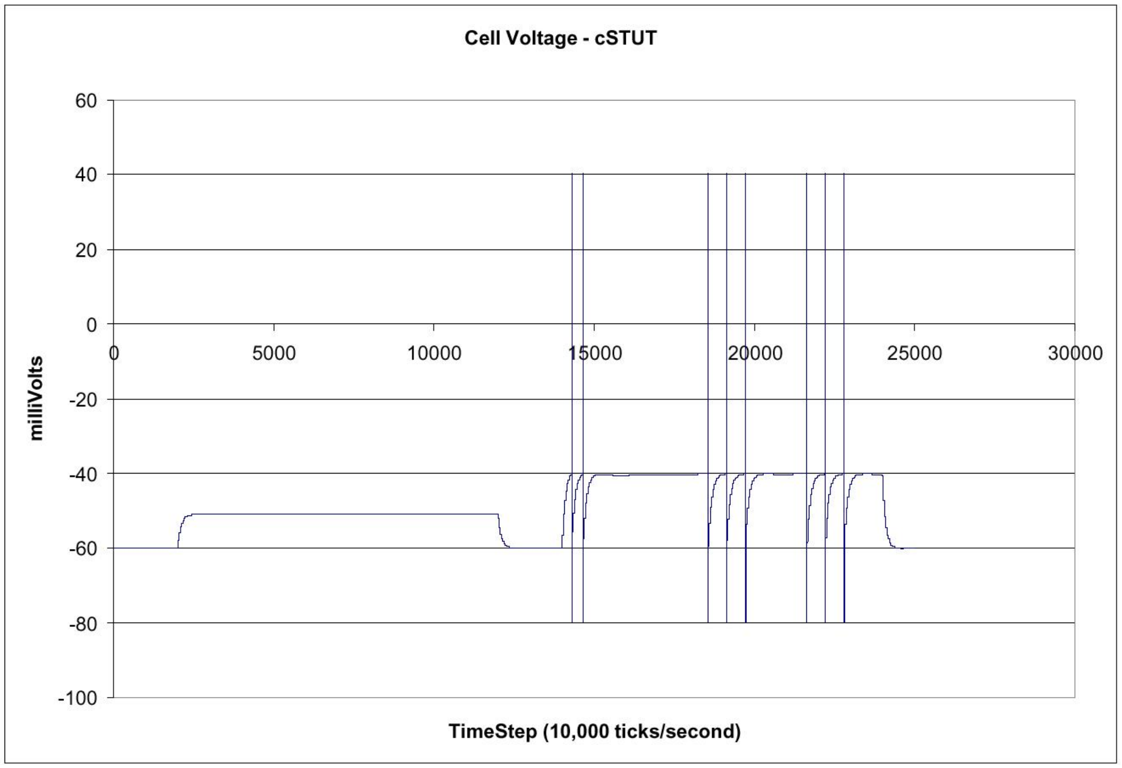

Figure 4.18: Cell Voltage with classic stuttering channel (cSTUT) [33].

The SIK channel shows the h and m particles interacting, and that the half-min values are set so that the effectiveness of the channel occurs after threshold has been reached.

4.2.3 Auditory Processing

Another test case of the simulator is the spike encoding of an auditory word as spoken by the author. The word spoken was “About” and it was saved as a text file of various decibels listed at specific frequencies across the time axis. This auditory file is first converted from decibels to a normalized sequence of numbers, then to a user specified frequency, then to a spike probability matrix, and then finally transformed into stimulus objects which fire only at a specific times and to a predetermined set of cells. For more information on the results of auditory processing in our simulator, please see[7].

In Figure 4.19, we show the audio (decibel) display of the word "About." The three distinct phonemes (/A/, /BOU/, /T/) can be seen in this image. Figure 4.20 shows the spike encoding of this word in approximately 1300 cells that received this input. Because the word "About" is spread out over time (approximately 0.5 seconds), this figure shows the spike activity of the cells over that 0.5 second interval. Figure 4.21 shows a density plot of the same spike encoded word. The three phonemes can be seen in timesteps [1..1,000] (/A/), timesteps [2,000..3,500] (/BOU/), and timesteps [4,800..5,000] (/T/).

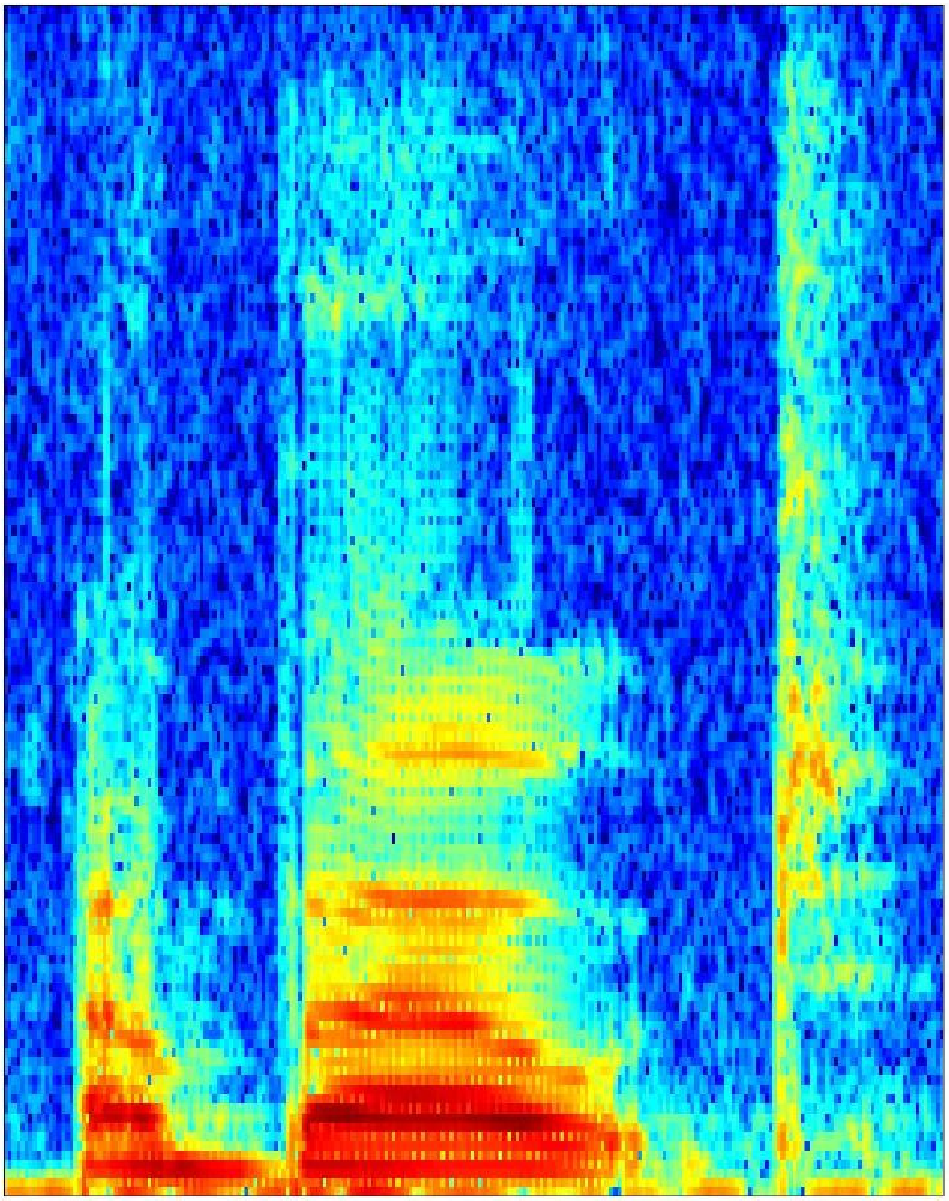

Figure 4.19: MatLab Output of spoken word “about” [34].

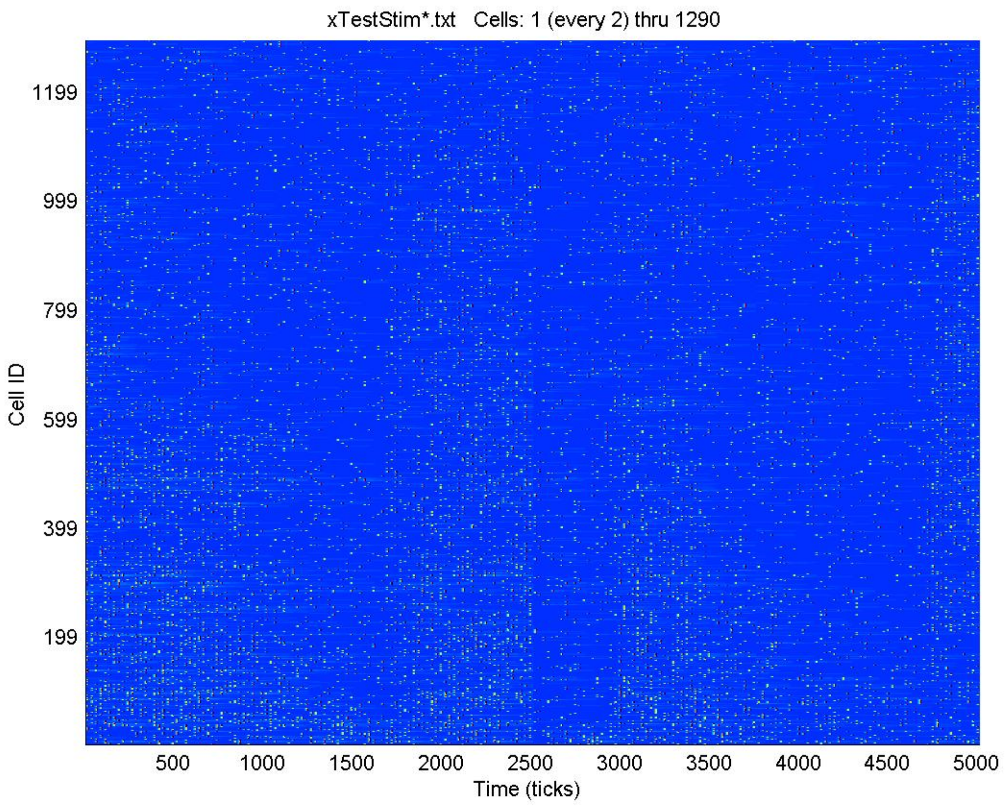

Figure 4.20: Spike Encoding of of spoken word “about” [34].

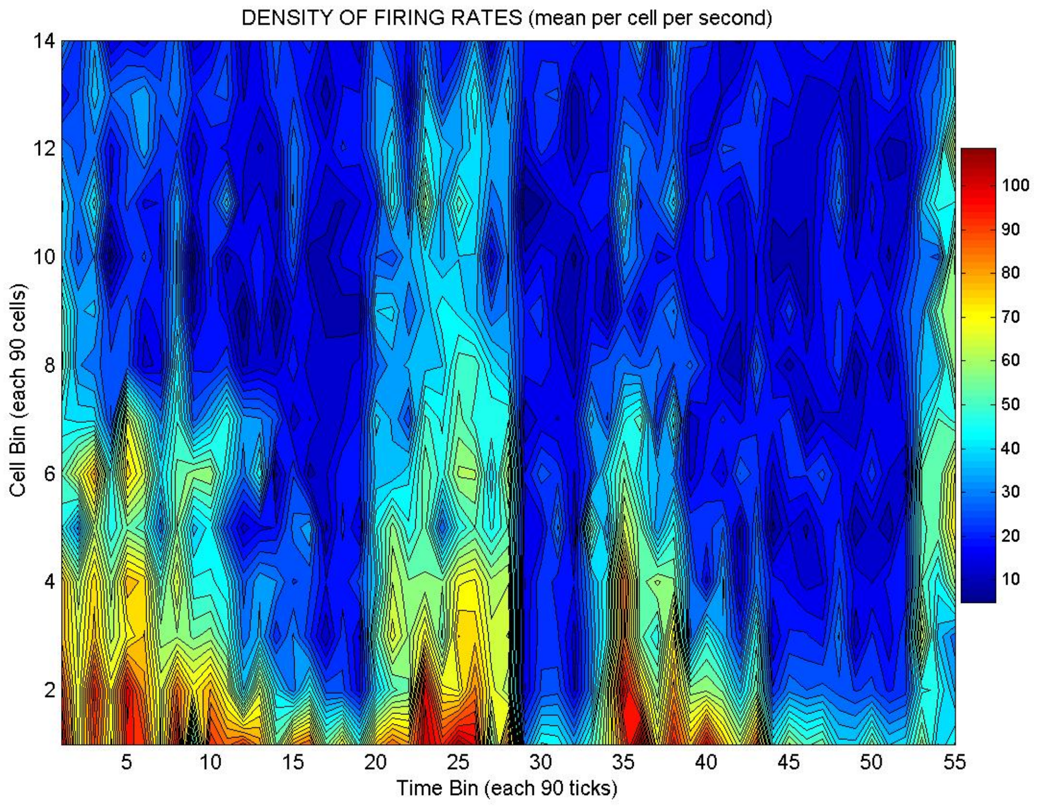

Figure 4.21: Density plot of spike encoded spoken word “about” [34].

Chapter 5

Conclusions and Future Work

5.1 Conclusions

We have shown that a very large-scale model of a mammalian neocortex can be designed and implemented using object-oriented techniques, static templates in key areas (spike and PSG waveforms), and the calculating of important biomechanics during the simulation (synaptic, channel and compartment dynamics). An in depth knowledge of the neocortical dynamics was necessary to accurately design and implement this simulator. The choice of object-oriented techniques for this task was made specifically to enable us to model the brain in a generic fashion. In this way, no parameters were hard coded, and all functionality was encapsulated within each object. Just as the biological brain modifies certain parameters while replicating functionality throughout its various regions, our *in silico* model can actively simulate other areas of the brain merely by changing the input parameters.

The object-oriented design also facilitated the modularity of the system. For example, we were able to swap out multiple communication paradigms as needed merely by creating multiple instances of the MessageBus object, and recompiling with whatever paradigm we wished to employ at the time.

Most importantly, however, the object-oriented system has enabled us to model the relationships of neurons within a given cortical community. It is this aspect that distinguishes this model from other simulators. While other tools are excellent for modeling single-cell networks, this simulator is able to model very large scale networks of highly connected neurons. Since it is through the relationships between active neurons that learning and memory occur, this simulator may provide a tool for developing and testing of learning algorithms of communities of cells.

5.2 Future Work

This simulator is far from complete, however, and in the following subsections, there is a list of possible directions for future work. This work will only enhance the system's capabilities, and in some cases may even provide more effective artificial intelligent classifications.

5.2.1 Multi-Compartmental Design

Each cell is comprised of multiple compartments, including at least one soma (minimum required amount for a point-to-point neuron model), multiple dendrites (apical and basal) and an axon. Each is designated by its [x,y,z] coordinates within the cell itself, with the soma being at ground zero [0,0,0] within the cell. A positive coordinate means it is stemming away from the cell body (in the direction of the output axon) and a negative coordinate means it is stemming away from the cell in the direction of the input dendrites.

These coordinates are used in conjunction with the speed of connection to calculate the compartment delay of passing a message from one compartment to another. This models the geometry of the cell itself, so that a dendrite that is very far away from the soma (in relation to the other compartments) will have a longer time in delivering its message to the soma. Each compartment connection has a conductance associated with it, and this is used to calculate the current from that compartment, and used to aggregate with other values to the net current within the soma, and to determine whether threshold has been reached.

Furthermore, in accomplishing synaptic algorithms with a multi-compartment design, it is important for each compartment to keep track of the intra-cell compartment from which it is receiving inputs. This will allow a compartment to update it's synaptic strengths when any post-synaptic compartment fires an action potential

5.2.2 Fuzzy Clustering and Distribution Algorithm

Since connectivity drives everything from memory utilization to communication speed, it is important to localize highly connected cells onto one node, and distribute more sparsely connected groups across node boundaries. A fuzzy clustering algorithm [27] based on nearest neighbor[10] and connectivity could be employed to aggregate and distribute cells. This algorithm would also take into account the maximum computational load on a given node, and utilize as many or as few nodes as needed.

5.2.3 Comparative of Biological Neural Network with Artificial Intelligent Systems in Speech Recognition Accuracy

It is important to test the BNN along side other forms of AI based classifiers. Hidden Markov Models (HMM) [31], Genetic Algorithms (GA) [17] and Artificial Neural Networks (ANN) [31] that are currently employed for speech recognition[31], have not been able to achieve greater than 95% accuracy in recognition. It is the goal of this simulator to utilize biological principles in organizing a column in the auditory cortex to generate discrimination among spoken words. This simulator must be able to recognize words against background noise.

5.2.4 Fault Tolerance

A large cluster can handle a simulation running for several hours, days or weeks. Fault tolerance and recovery is important for maintaining data integrity and conserving research time. As a result, a separate object could be established to monitor the health of each node and to shift the data and processing off of an unhealthy node to other nodes.

5.2.5 Additional Channel Dynamics

Currently the simulator models only potassium (K^+) channels, and mimics a Sodium channel using K^+ channel equations. There are many other channels in existence within a biological system, and these would need to be modeled in order to more accurately represent a cortical column [35].

5.2.6 Long Term Memory Dynamics

Currently, synaptic dynamics are modeled using equations for Redistribution of Synaptic Efficacy, positive and negative Hebbian Learning. Each of these processes modifies the `U_mean` available in the synapse based on unchanging time constants. Current research is showing that these time constants are not static, but dynamic. This change would add another level of complexity into the synaptic dynamics [21].

Bibliography

[1] James M. Bower and David Beeman. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* Springer-Verlag New York, Inc., 2nd edition, 1998.

[2] Javier Campos, Susanna Donatelli, and Manuel Silva. Structured solution of asynchronously communicating stochastic modules. *IEEE Trans. on Soft. Engr.*, 25(2):147–165, March-April 1999.

[3] Satish Chandra, Bradley Richards, and James R. Larus. Teapot: A domain specific language for writing cache coherent protocols. *IEEE Trans. on Soft. Engr.*, 25(3):273–278, May-June 1999.

[4] Stephen R. Deiss, Rodney J. Douglas, and Adrian M. Whatley. *Pulsed Neural Networks*, chapter A Pulse-Coded Communications Infrastructure for Neuromorphic Systems. MIT Press, Cambridge, MA, 1999.

[5] Behrouz A. Forouzan. *Data Communications and Networking.* McGraw Hill Higher Education, 2nd edition, 2001.

[6] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine – A User's guide and tutorial for networked parallel computing.* MIT Press, Cambridge, MA, 1994.

[7] Philip H. Goodman, E. Courtenay Wilson, James B. Maciokas, Frederick C. Harris Jr., Sushil Louis, Anirudh Gupta, and Henry Markram. Large-scale parallel simulation of physiologically realistic multicolumn sensory cortex. under review, June 2001.

[8] Willaim Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

[9] Anirudh Gupta, Yun Wang, and Henry Markram. Organizing principles for a diversity of GABAergic interneurons and synapses in the neocortex. *Science*, 287:273–278, January 14, 2000.

[10] John A. Hartigan. *Clustering Algorithms.* John Wiley and Sons, NY, Inc., New York, 1975.

[11] M.L. Hines and N.T. Carnevale. The NEURON simulation environment. *Neural Computation*, 9:1179–1209, 1997.

[12] M.L. Hines and N.T.Carnevale. Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation*, 12:839–851, 2000.

[13] Dax A. Hoffman, Jeffery C. Magee, and Costa M. Colbert. K^+ channel regulation of signal propogation in dendrites of hippocampal pyramidal neurons. *Nature*, 387:869–875, 1997. Correction in volumn 390, pg 199.

[14] F.W. Howell, J. Dyhrfjeld-Johnsen, R. Maex, N. Goddard, and E. De Schutter. A large-scale model of the cerebellar cortex using pgenesis. *Neurocomputing*, 32-33:1041–1046, 2000.

[15] Christof Koch. *Biophysics of Computation.* Oxford Univ. Press, New York, NY, 1999.

[16] Christof Koch and Idan Segev, editors. *Methods of Neuronal Modeling.* MIT Press, Cambridge, MA, 2nd edition, 1998.

[17] John R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection.* MIT Press, Cambridge, MA, 1993.

[18] Andreas Kreusch, Paul J. Pfaffinger, Charles F. Stevens, and Senyon Choe. Crystal structure of the tetramerization domain of the shaker potassium channel. *Nature*, 392:945, 1998.

[19] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[20] Wolfgang Maass, Thomas Natschlager, and Henry Markram. Computation in neural microcircuits: A new approach based on perturbations. under review, June 2001.

[21] Henry Markram and Anirudh Gupta. Unpublished data, June 2001.

[22] Myricom Inc. Creators of Myrinet. `http://www.myrinet.com`, July 2001. 325 N. Santa Anita Ave. Arcadia, CA 91006.

[23] Peter Pacheco. *Parallel Programming with MPI.* Morgan Kauffman, San Francisco, CA, October 1996.

[24] William H. Press, Saul A Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, MA, 2nd edition, 1992.

[25] Dale Purves, George J. Augustine, David Ditzpatrick, Lawrence C. Katz, Anthony-Samuel LaMantia, and James O. McNamara, editors. *Neuroscience.* Sinauer Associates, Inc., Sunderland, MA, 1997.

[26] Michael Reece. *Pulsed Neural Networks*, chapter Encoding Information in Neuronal Activity. MIT Press, Cambridge, MA, 1999.

[27] P.J. Rosseeuw, E. Trauwaert, and L. Kaufman. Fuzzy clustering with high contrast. *Journal of Computational and Applied Mathematics*, 64:81–90, 1995.

[28] Manuel A. Sanchez-Montanes, Peter Konig, and Paul F.M.J. Verschure. Learning in a neural network model in real time using real world stimuli. *Neurocomputing*, 38-40:859–865, 2001.

[29] Walter Senn, Henry Markram, and Misha Tsodyks. An algorithm for modifying neurotransmitter release probability based on pre- and post-synaptic spike timing. *Neural Computation*, February 2000.

[30] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongerra. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, 1996.

[31] Edmondo Trentin and Marco Gori. A survey of hybrid ANN/HMM models for automatic speech recognition. *Neurocomputing*, 37:91–126, 2001.

[32] Misha V. Tsodyks and Henry Markram. The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. *Proc. Natl. Acad. Sci. USA*, 94:719–723, January 1997.

[33] Displayed using channel data generated by James B. Maciokas, July 2001.

[34] Displayed using MATLAB program written by Philip H. Goodman, July 2001.

[35] Walter M. Yamada, Christof Koch, and Paul R. Adams. *Methods in Neuronal Modeling*, chapter Multiple Channels and Calcium Dynamics. MIT Press, Cambridge, MA, 2nd edition, 1998.