Implementation of a Biologically Realistic Parallel Neocortical-Neural Network Simulator*

***E. Courtenay Wilson*†, *Philip H. Goodman*‡, *and Frederick C. Harris, Jr.*§**

1 Introduction

The primary goal of this simulator is to create a novel classifier based on a biologically realistic neocortical-neural network. Parallel processing of this very large-scale, object-oriented simulator is key for approaching real-time simulation of synaptic and neocortical network dynamics. Clustering algorithms applied to the dense cell-connection matrix enable load-balancing and data parallelism by organizing highly connected groups of cells onto a particular node, thus reducing the performance cost of inter-nodal communication.

The simulation is accomplished by modeling a whole community of cells within a brain structure and observing the emergent behavior of this system. This modeling is done using fine grain parallelization, as opposed to very small scale cellular networks and coarse grain parallelism, such as that found in NEURON or GENESIS[7].

The structure of the rest of this paper is as follows: in Section 2 we begin with an overview of the connectivity within the brain and its relevance to our simulation. In Section 3 we present the Object-Oriented design of this simulation. Section 4

*Funding for this paper furnished by the Office of Naval Research under multiple contracts.

†Department of Computer Science, University of Nevada, Reno, ecwilson@cs.unr.edu

‡Department of Internal Medicine, University of Nevada School of Medicine, Reno, goodman@unr.edu

§Department of Computer Science, University of Nevada, Reno, fredh@cs.unr.edu

follows with an overview and a discussion of how the control flow is implemented, including MPI's role in internodal communication Section 5 wraps up this paper with our Results, Conclusions, and a brief discussion of some Future Work.

2 Biological and Computational Motivations

2.1 Biological Topology

The computational topology of the cortical simulator is based on a biological model of a mammalian neocortex. The neocortex is organized into functional units called columns, which are made up of multiple layers. Each column and each layer perform given tasks and contain highly connected groups of various cell types.

The cell types that make up the columns and layers of the neocortex are heterogeneous and perform various functions. For example, some cell types contain excitatory synapses, which means that their synaptic outputs will stimulate the connected cell to raise its voltage level, thus bringing it closer to firing an action potential. Other cell types, called *interneurons*[4], contain inhibitory synapses, and their outputs typically have a dampening effect on connected cells, lowering the voltage of the connected cells, and making it harder for the connected cell to fire an action potential.

Each cell type consists of compartments[6], such as dendrites, a soma, an axon, channels[5], and synapses. The functions of each are roughly described as follows: The dendrites are responsible for bringing inputs into the soma, these inputs typically come from other cells that have reached threshold and are firing an action potential. The soma is where the integrate-and-fire process takes place–a process which entails the aggregation of all inputs, calculation of membrane voltage, and the decision of whether or not to fire an action potential. The axon is responsible for carrying the output signal to the cells that will receive this signal.

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 to a resulting analog *post synaptic current* (PSC). The synapses are classified as either excitatory or inhibitory. The excitatory PSC is a positive waveform, and the inhibitory PSC is a negative waveform.

The channels are responsible for affecting the membrane voltage by accepting or rejecting certain ions. These ion fluxes are triggered by the release of neurotransmitters from the connecting synapse. Certain types of channels will help a cell reach threshold and fire, and other types of channels will dampen this response.

The mechanics of the integrate-and-fire process are such that a given compartment within a cell aggregates all of its inputs, calculates the resulting membrane voltage, and, if it is equal to or greater than the threshold for that compartment, outputs a resulting spike shape as the action potential. The spike[8] shape is a short burst of voltage, on the order of only seven to ten milliseconds, and ranging from the threshold value to somewhere above positive 40 millivolts. This spike shape triggers the synapses on the connecting cells to release neurotransmitters. This release results in a PSC waveform on the connecting compartment's membrane, thereby beginning anew the cycle of a binary event (the spiking action potential) triggering

an analog event (the PSC), which triggers a binary event, and so on.

2.2 Computational Topology

The computational topology is closely modeled after the biological principles mentioned above. The user specifies a template for each segment of the model, with biological principles specified. The segments are labeled as follows: Anatomical Elements (column and layer shell); Physiological Elements (stimulus, channel, synapse, spike shapes); Physiological Constructs (compartment, cell type); Anatomical Constructs (layer, column, brain); and Reports.

Certain biomechanics are mimicked through templates rather than an intricate modeling process. For example, the spike shape and PSC waveforms are two such templates that are specified by the user. The choice for making some processes into templates was done to expedite the modeling of very large scale networks. Conversely, other neuron modeling tools, such as NEURON and GENESIS, model these processes explicitly in single neuron models. Thus one has the trade off between network size and computational complexity.

The cell/compartment to cell/compartment connectivity is specified by the user as a probabilistic function, where only the compartments within a given cell are connected absolutely. For example, within layer 4, cell_type1 is connected to cell_type2 by synapse_type1 and probability 0.5. Connectivity on all levels of the brain can be specified, or left blank. If no connectivity is specified, the program will run, but no communication between cells will take place.

3 Object-Oriented Design

Because the brain itself is segmented into compartments and individual objects such as synapses, dendrites, and channels, the choice of the object-oriented design matches the biological model. Each object utilizes the Standard Template Library as containers for other objects, and each object encapsulates functionality that is specific to that object only. In this way, the synapse object handles synaptic learning, outputting of PSC waveforms, and Redistribution of Synaptic Efficacy calculations[10]. Likewise the channel object encapsulates the calculations of channel currents and outputs those currents to the parent compartment.

The object-oriented model allows the simulator to model objects generically, changing their results through the input parameters without affecting the underlying object functionality. In this way, we can model other areas of the brain with little computational changes, changing primarily the input parameters.

On a computational level, there are several basic forms of objects. Some are containers for other objects and maintainers of administrative data, while others are both containers and state machines. Other objects perform specific functions that are important to the computational side but do not directly effect the biological system.

The fundamental types consist of the following objects: synapses, channels, and spike shapes. These are owned by another object, perform specific biological or data-collection functions, and receive input from and output data to their owning

object. The spike shape is responsible for giving to its owner object the values for a given indexed timestep. These values mimic the spike shape that a certain compartment may output after reaching threshold and firing.

The container objects are ones that are holders for other objects and, in some cases, for administrative data. These objects are responsible for ensuring that each owned object is visited and updated. One of the container objects is the cell object, which is a container for the compartment objects. It is responsible for ensuring that the messages are delivered to the appropriate compartment, whether that message comes from outside the cell or within the cell (from another compartment). Of these contained compartments, only the soma is the minimum required to mimic a point to point model.

Another container object is the brain itself. The brain is a container for the virtual global cells list, the stimulus objects, and the report objects. The brain is responsible for ensuring that the stimulus objects are visited (to send data to the cells), that the report objects are visited (to collect data on the cells), that the MessageBus object is visited (to ensure message passing), and that each constituent cell/compartment is visited. The brain object is also responsible for such administrative duties as calculating for how many time ticks the program runs and incrementing the counter on each time tick.

The object that is both a container and a state machine is the compartment object. It is a container of such objects as synapses, channels, and spike shapes. It is a state machine in that its membrane voltage is calculated and updated at each time step. This is where the heart of the cortical simulator is located. All the integrate-and-fire routines exist here, and all action potentials are fired from compartment objects to other compartment objects (within cell or outside cell). If there is no input from the contained objects or external sources, the compartment still changes state–it's membrane voltage degrades by a pre-set amount towards resting potential.

The compartment is a generic object that switches based on given input parameters. For example, only one compartment object is used to model all of the compartments within the cell–that is the basal dendrites, the apical dendrites, an axon, and the soma. Everything is based on the variables that were specified in the user input file, including whether or not this particular compartment will fire an action potential, and, if so, which spike shape it will utilize.

The compartment also contains a list of cell/compartment data, to which cell/compartment it is sending action potentials, and from which cell/compartment it is receiving action potentials. This connection matrix forms the heart of the inter-cell/compartment connectivity and message passing.

The other types of objects are those that perform specific functional tasks. The stimulus object calculates a given voltage or current stimulus based on the user-defined choice of types ranging from a linear function, to a pulsed (staircase or constant) function, to a sine function. These functions are designed to mimic signals that are coming in from other parts of the brain as well as to mimic external stimuli (like a voltage clamp).

The report object handles the data processing of what to report on, which cell/compartment to receive this report, and at what frequency the report is to be

received. The most detailed reporting is done every single timestep, although this is computationally expensive.

The FileIO object handles the file input/output in a threaded manner. It outputs the data from a cell/compartment to a given file. This output file is later processed and reformatted into a more user friendly display.

The message object is a collection of data that is used in communication between all objects within the simulator. The ownership of the message object (once instantiated), passes from the sender to the receiver. It is the responsibility of the sender to allocate the memory for the message and for the receiver to deallocate the memory once used.

One of the most important functional objects is the MessageBus object. Its detailed model will be discussed in a later section; however, it is responsible for all of the message passing between cells, stimulus and cells, and reports and cells. It encapsulates all of the MPI calls, and, through MPI, it is responsible for all internodal 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.

4 Simulator Implementation

4.1 Overview

In the brain, there is a high degree of connectivity within columns of highly clustered cells and much less connectivity across column boundaries. The input data file mimics the biological realism of the brain as much as possible by stipulating connectivity on several levels: intra-cell (inter-compartment), intra-layer (inter-cell), intra-column (inter-layer), and intra-brain (inter-column)[2]. The connectivity of cells within a layer, column, or brain is specified by the user as a probabilistic function. Only the intra-cell connectivity is non-probabilistic: a cell either has a specific compartment or it doesn't.

The parallel algorithm for the cortical network simulator consists of several steps. The first step is the reading and parsing of the input data file. Once all of the data templates are input and error-checked for biological accuracy, the distribution begins. The distribution algorithm makes a connection matrix based on the connectivity specified in the input data file. A deterministic clustering algorithm is then employed to aggregate groups of high connectivity.

If a cell is considered "empty" (meaning it is not connected to other cells) and the user has specified the ignoring of empty cells, then it is deleted, and the connectivity matrix is modified to reflect this change. This modification improves the load of computation by ignoring those cells that do not contribute to the overall network of cells within the brain, while mimicking the biological realism of the phase of cell die-off. The cell data are then partitioned so that each node in the cluster has its own portion. This reduces the performance cost of inter-nodal communication by grouping together highly connected cells onto one node.

Once the cell distribution map is established, each node instantiates the objects according to the global connection matrix if that object has been designated

for that node. The individual cell's connection matrix is created and owned by that cell. Now the program is ready to begin the actual neocortical simulation.

The main simulation consists of a brain loop in which all the cell-to-cell communication is handled via a MessageBus object[1], which functions as a router for packing and delivering of the messages. This structure allows us to encapsulate the message-passing paradigm, thus allowing us to swap out our current MPI-based implementation with other communication paradigms. This is a critical issue in porting of this simulator to other hardware platforms.

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, the receiving address, and time step (i.e., the message envelope). The dynamic data set contains such information as the membrane voltage values, external stimuli values (current or voltage), or reporting information.

When a cell fires an action potential it communicates this event through the MessageBus to all the cells in its connection matrix. The MessageBus then relays these messages to the cells specified. If the receiving cell is off-node, it is the responsibility of the MessageBus to package this in an appropriate way and communicate it to the proper node, at which point the MessageBus on that node forwards it to the local recipient cell. The sending and receiving of messages is done in such a way as to prevent node starvation or deadlock. These new messages are then buffered to be read by the MessageBus on that local node.

This whole process occurs for each simulated time tick and is synchronized to prevent a race condition whereby certain nodes can become many time ticks ahead of other nodes in the simulation. The synchronization is important for maintaining biological realism as it allows each cell to aggregate all inputs for each time tick in order to perform its integrate and fire calculations.

4.2 Control Flow Implementation

The control flow of the cortical simulator consists of three major stages: initialization, brain computations, and finally post-processing of the data. Each stage and its constituent parts are discussed in some detail within this section.

The initialization stage is handled by the InitializationManager object and consists of multiple functional units. This stage occurs on all nodes within the cluster concurrently with switches for computation that should occur on particular nodes.

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, 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 the Standard Template Library vectors.

The second step of initialization is the displaying of values to the user. The DisplayObjects 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, it will display the incorrect variable along with the correct range of values for that variable.

The fourth step in initialization is 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. This object holds the groups of cell types, their location in the brain (column & layer), and their global cell id values. The CreateConnectMatrix object updates each cell list in the CellManager with the constituent cells connections.

The fifth step creates the DistributionManager object which uses the connection information now stored in the CellManager to compute the distribution of the 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 when instantiating the cell objects.

The sixth step in initialization is the instantiating of each object from the biological template provided by the user. The MakeObjects object handles this task. Each node in the cluster has a MakeObjects object which is responsible for checking the current machine number, and only making those cells which are situated on that particular machine. The CellManager object is utilized heavily for this task as it stores the machine location for 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 the different nodes within the cluster.

The last step in initialization is the connecting of the cell/compartments. This is handled on each node, where each node fills in the connection matrix for the given cell/compartment based on whether or not it exists on this node. The Synapse objects are then instantiated on the receiving node, and they are then owned by the receiving cell/compartment. The MessageBus is employed at this point to send data to each cell/compartment with the synapse ID values.

After the initialization stage is complete, each constituent helper object is deleted. The brain objects on each machine are synchronized before beginning the next stage of the simulation, which is the brain computation. This computation is the same on each node. On each timestep, the brain visits the objects that it contains. These include the stimulus objects (which may or may not send values to cell/compartments), the reports objects (which may or may not send report requests to cell/compartments), the MessageBus (to send and receive messages and pass them along to cell/compartments on this node and on external nodes), and each cell within its local list. These cells then visit each compartment contained within them, which process the integrate-and-fire routines mentioned above.

The third and final stage of the simulator takes place once the Brain object has completed its pre-determined number of cycles. This stage consists of using the PostProcessing object to reformat the output data files into a more user friendly context. This process inputs each output file and reformats it into a specified display which could be either a text file or a web-based PHP file. Once this post processing is

completed, the simulation deletes all allocated memory and the program terminates.

4.3 MPI and Communication

The MessageBus object is an important object for facilitating communications between objects within each brain. Through the use of MPI[3, 9, 11], our current implementation handles communication with external nodes. The communication scheme of the cortical simulator is best described by the diagram in Figure 1.

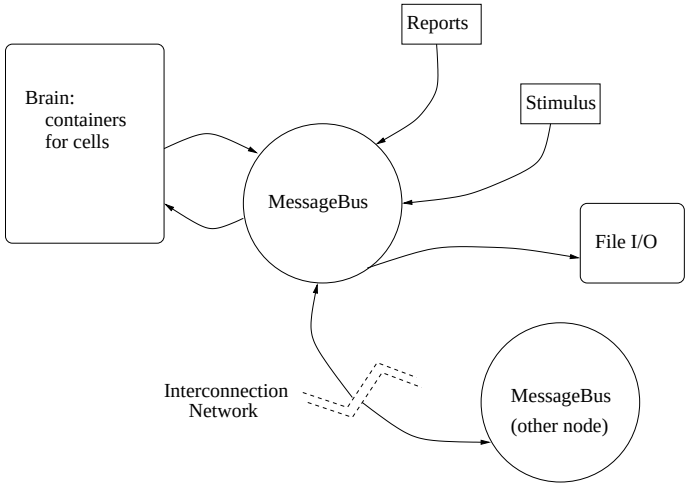

Figure 1. *Communication Model*

The algorithm for preventing deadlock and starvation is based on a simple All-to-all personalized message exchange 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 for each node within 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 uses blocking sends (MPI_Send) and blocking receives (MPI_Recv).

In order to send a message to an off-node cell, the MessageBus processes the outgoing message queue and formats each message to a particular node into a user defined MPI data type, stores it onto a user defined MPI buffer, and sends this buffer to the designated destination node. Once the message has been sent, the memory allocated by that message object is deleted.

On a local level, the MessageBus checks the destination of each message, and if it is the same as the current node, no MPI communication operations are used for the message object to be passed to the particular cell/compartment addressed in the message.

The MessageBus also uses a 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).

5 Results, Conclusions, and Future Work

The sequential implementation was finished at the end of the summer of 2000. Once this implementation was completed, it was evaluated and tested for biological realism and accuracy. This was accomplished by comparing our results with published and accepted results for channels[5], compartments[7], and synapses[4, 10].

The sequential implementation was then changed into a parallel implementation as described previously, and we are in the process of evaluating this implementation. At this point in our research, we have tested our implementation on two architecture platforms. The first platform is a dual processor Sun Enterprise server with 2GB of shared memory, and the second platform is a Beowulf cluster of 8 Pentium II 400 machines with dual 100 Mbs Ethernet connections to a Bay network switch.

On the shared-memory architecture, we have run the simulation with one and two processors and have scaled the number of cells per processor from 40 to 200. On the Cluster we have run the simulation with one, two, and four processors and have also scaled the number of cells from 40 to 200 per node.

Connectivity in each of these cases was set at such a level that would stress the implementation and help us determine the architectural limits of each hardware platform. For our tests we chose a cell connectivity of approximately n^2. Seventy-five percent of the cells were receiving external voltage from the Stimulus object which was set at such a level so as to force an action potential to occur. This stimulus arrived at regular intervals for 60% of the total simulation time. When the cell receives this stimulus and fires an action potential it then sends a message to each cell to which it is connected which, in turn, causes a cascade of communication.

In Figure 2 we show the results of our simulation running on the dual processor Sun shared-memory machine, and in Figure 3 we show the results from the Beowulf cluster using the same number of CPUs and cells.

It is clear from comparing these results that communication latency is a major factor in performance of our simulation. We have shown an improvement in the performance on a well-connected machine but a decrease on a loosely coupled machine. Therefore, in order to achieve the results we desire, we will need to continue evaluation on an architecture that can support massive communication between the cells.

In conclusion, once this simulator is operating with hundreds of thousands to millions of cells, it will allow us to address the following types of questions: What minimal micro-circuit must be replicated to create a functional cortical column? How many such columns must interact to demonstrate emergent behavior, such as the remarkable generalization ability of mammalian brain "classifiers"? We will also be able to compare brain-like computation to existing artificial neural network and traditional non-neural classifiers in order to address several other challenging

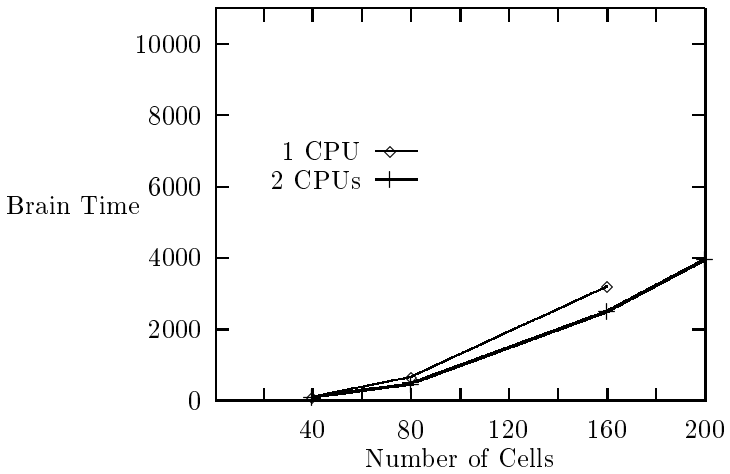

Figure 2. *Brain Time (in seconds) on a dual processor shared memory machine*

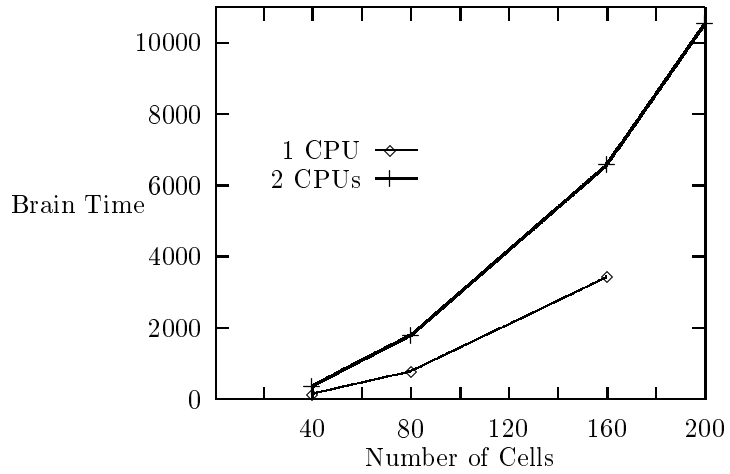

Figure 3. *Brain Time (in seconds) on our Beowulf Cluster*

problems such as pattern recognition, real-time human gesture recognition, and navigation.

Bibliography

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

[2] Satish Chandra, Bradley Richards, and James R. Larus. Teapot: A domain-specific languages for writing cache coherence protocols. *IEEE Trans. on Soft. Engr.*, 25(3):pp. 317–333, May/June 1999.

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

[4] 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.

[5] Dax A. Hoffman, Jeffery C. Magee, Costa M. Colbert, and Daniel Johnston. K^+ channel regulation of signal propogation in dendrites of hippocampal pyramidal neurons. *Nature*, 387:869–875, June 26 1997. Correction in volume 390 pg 199.

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

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

[8] Wolfgang Maas and Christopher M. Bishop, editors. *Pulsed Neural Networks*. MIT Press, Cambridge, MA, 1999.

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

[10] Walter Senn, Henry Markram, and Misha Tsodyks. An algorithm for modifying neurotransmitter release probability based on pre- and post-synaptic spike timing. *Neural Computation*, to appear. Accepted February 16, 2000.

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