University of Nevada
Reno

# Parallel Optimization of a NeoCortical Simulation Program

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

by

James Frye

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

December 2003

We recommend that the thesis prepared under our supervision by

**James Frye**

entitled

**Parallel Optimization of a NeoCortical Simulation Program**

be accepted in partial fulfillment of the requirements for the degree of

**MASTER OF SCIENCE**

---

Dr. Frederick C. Harris, Jr., Ph.D., Advisor

---

Dr. Angkul Kongmunvattana, Ph.D., Committee Member

---

Dr. Phillip H. Goodman, M.D., At-Large Member

---

Marsha H. Read, Ph.D., Associate Dean, Graduate School

December 2003

qq

## Abstract

This thesis describes work done in optimizing an existing NeoCortical Simulation Program (NCS), including the development of a set of parallel profiling and measurement tools.

The NCS program is an ongoing project of the Brain Computation Lab. Previous development work was most recently presented in [18]. Using the results presented there as a baseline, it will be shown that this work has increased computation speed by at least an order of magnitude; increased the demonstrated model size by three orders of magnitude; created a program which exhibits near-linear speedup over the number of processors available for testing; and, despite having added significant additional functionality, has decreased the code base by some 45 percent.

**Acknowledgements**

Thanks are due to

- Dr. Phillip Goodman, who conceived the NCS project, did the early research, founded the Brain Computation Laboratory at UNR, does the bulk of the administrative and funding work, and who continues to be the primary researcher on the biomedical side.
- Dr. Fred Harris, who organizes both hardware and software support.
- Jason Baurick and Lance Hutchinson for system support.
- Rich Drewes and Jim King for help with the coding.
- James Maciocas, Jake Kallman, Matt Ripplinger, and Christine Wilson for uncounted hours of testing.
- Cindy Harris, for proofreading.
- Harlie, for hikes, ball chasing, playing in the snow, and proving that old dogs *c*an learn new tricks.

# Contents

**Abstract** **i**

**Acknowledgements** **ii**

**List of Figures** **iv**

**List of Tables** **v**

**1 Introduction and Motivation** **1**
1.1 History . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.3 Terms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Necessary Biology . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.5 Hardware . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**2 Overview of Program Design and Operation** **7**
2.1 The Input File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Initialization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Input and Parsing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.4 Global Index Creation . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.5 Distribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.6 Brain Construction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.7 Connection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.8 Stimulus and Report Creation . . . . . . . . . . . . . . . . . . . . . . 11
2.9 Thinking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.10 Internal Cell Processing . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.11 Compartment Processing . . . . . . . . . . . . . . . . . . . . . . . . 13
2.12 Channels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.13 Synapse and Spike Processing . . . . . . . . . . . . . . . . . . . . . 15
2.14 Synapse Learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

**3 Profiling Tools** **18**
3.1 Internal Profiling Routines . . . . . . . . . . . . . . . . . . . . . . . 19

3.1.1 Profiling Execution Time . . . . . 19
3.2 The QQ Profiling Library . . . . . 20
3.2.1 Basic Theory of Operation . . . . . 21
3.3 Profiling Memory Use . . . . . 22
3.3.1 Object Creation . . . . . 24
3.4 External Monitoring Programs . . . . . 24
3.4.1 `memsnoop` . . . . . 25
3.4.2 Parallel `gdb` . . . . . 25

**4 Improving Parallel Performance** **27**
4.1 Load Balancing . . . . . 27
4.1.1 Load Balancing Algorithm . . . . . 29
4.1.2 Memory Balancing . . . . . 31
4.1.3 Implementation . . . . . 32
4.2 Connections . . . . . 32
4.2.1 Creating Connections . . . . . 32
4.2.2 Making Connections . . . . . 33
4.3 Message Passing & Synchronization . . . . . 35
4.3.1 The Message-Passing Algorithm . . . . . 37
4.3.2 Synchronization . . . . . 39

**5 Results** **42**
5.1 Models Used In Testing . . . . . 42
5.2 Sequential Performance Improvements . . . . . 43
5.2.1 Parallel Performance . . . . . 47
5.2.2 Virtual Processors . . . . . 50
5.3 Memory Use . . . . . 51

**6 Conclusions and Future Work** **56**
6.1 Performance - Time . . . . . 56
6.2 Performance - Memory . . . . . 57
6.3 Analysis . . . . . 58
6.4 Future Work . . . . . 59
6.5 Finally... . . . . . 60

# List of Figures

1.1 Typical Spike Profile. . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1 Typical Post-Synaptic Conductance (PSC) Template. . . . . . . . . . . . . 16

3.1 Plot of Typical QQ Output. . . . . . . . . . . . . . . . . . . . . . . . . 22

4.1 Connection Times. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.2 Structure of Message Packet. . . . . . . . . . . . . . . . . . . . . . . . 37
4.3 Schematic of MessageBus. . . . . . . . . . . . . . . . . . . . . . . . . 38
4.4 Idle Time Due to Load Imbalance. . . . . . . . . . . . . . . . . . . . . 41

5.1 Share of CPU Time Used by Functional Areas, 1Column Model. . . . . . . 46
5.2 Share of CPU Time Used by Functional Areas, IVO Model. . . . . . . . . . 47
5.3 Processing Time and Firing Rate . . . . . . . . . . . . . . . . . . . . . 48
5.4 Variation of Execution Time with Spike Rate. . . . . . . . . . . . . . . . 49
5.5 Parallel Run Times, IVO with No Firing. . . . . . . . . . . . . . . . . . 50
5.6 Parallel Run Times, IVO with Firing . . . . . . . . . . . . . . . . . . . 51
5.7 Parallel Speedup for IVO Model. . . . . . . . . . . . . . . . . . . . . . 52
5.8 Run Times for BigIVO Model . . . . . . . . . . . . . . . . . . . . . . . 53
5.9 Run Times for AVI Model . . . . . . . . . . . . . . . . . . . . . . . . 54

# List of Tables


1.1 Myrinet/MPI Packet Times. . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

4.1 Compute Factors . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.2 Example of Load Balancing. . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3 Example of Memory Balancing. . . . . . . . . . . . . . . . . . . . . . . . 31

5.1 Characteristics of Test Models . . . . . . . . . . . . . . . . . . . . . . . . 43
5.2 Preformance Ratios of Functional Areas. . . . . . . . . . . . . . . . . . . . 46
5.3 Virtual and Dual CPU Performance . . . . . . . . . . . . . . . . . . . . . . 51
5.4 Memory Used by Components - NCS3 *vs* NCS5 . . . . . . . . . . . . . . . . 55
5.5 Memory Use by AVI Model (MBytes). . . . . . . . . . . . . . . . . . . . . 55

6.1 Code Size Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

# Chapter 1

# Introduction and Motivation

The NeoCortical Simulation program (NCS) is a project of the Brain Computation Laboratory at the University of Nevada, Reno. This laboratory is an ongoing collaboration between the UNR Department of Computer Science and the University of Nevada School of Medicine, which is applying computer modeling to the investigation of the fundamental operation of the brain.

Despite a half century of progress in computer technology, the performance of the brain remains unrivaled at many tasks. For instance, primates can correctly classify environmental stimuli and respond to them within 100 milliseconds of presentation — a fact that is all the more surprising when it is realized that the brain is at base a very low-speed device. Typical pyramidal neurons, *in vivo*, fire at rates between 10 and 40 Hz [2], so only a few "machine cycles" must somehow process the entire complex of sensory, associative, and motor events that constitute perception and response. It thus appears that the mammalian cortex must operate on principles very different from those employed by conventional computers.

One hypothesis is that the brain encodes and processes information by pulse-coding – that is, by the timing of spikes among a population of neurons. It is this hypothesis that the Brain Computation Laboratory proposed to test by creating NCS: the first large-scale, synaptically realistic computational model of the mammalian cortex.

## 1.1 History

Preliminary work by Goodman [8, 17] served as the basis for an initial pilot project in 1999 that implemented a biologically realistic simulator in Matlab. The cells modeled in this prototype successfully learned to reproduce synchronous input-output activity across multiple-layered cortical regions without the need for explicit 'back-propagation or recurrent output-input interconnection. In general, very complex dynamics, including periodicity, oscillation, and chaos, could be replicated.

Between 1999 and the summer of 2001, the prototype software was redesigned using object-oriented design principles and recoded in C++ [18, 19, 20]. The principal goals of this phase were to increase the biological realism of the model, and to allow users to input brain designs and stimuli in a form directly related to the biology. Due to the size and computational demands of the problems the model was intended to study, this version was designed from the beginning to run in parallel.

In this design, a "brain" consists of objects such as cells, compartments, channels, and the like, which model the corresponding cortical entities. The cells, in turn, communicate *via* messages passed through synapse objects. Input parameters allow the user to create many variations of the basic objects in order to model measured or hypothesized biological properties.

This work produced a functional simulation program in which the basic biological components were developed and tested in a working sequential program. A preliminary parallel version had demonstrated the ability to run simulated "brains" containing up to 1.5 million synapses. Although this version allowed the lab to begin some of its planned research [11, 14, 16], it was somewhat disappointing in both compute speed and maximum model size.

## 1.2 Goals

This phase of the project set goals of increasing the maximum brain size by approximately three orders of magnitude (to $10^6$ cells and $10^9$ synapses) while decreasing execution time for a given model by at least an order of magnitude.

## 1.3 Terms

A number of shorthand terms commonly used in this thesis are defined here.

- Action Potential (AP) - A synonym for Spike.
- Brain - A set of inputs to one of the NCS programs, defining the cells and synapses that are to be simulated.
- Cluster - A group of identical cells, as specified by a `CELL_TYPE` input statement. The cluster is the basic address unit of the input file: connections are made between two clusters, stimuli go to specified clusters, and reports are specified by clusters.
- NCS3 - The version of the simulation program described in [18].
- NCS5 - The version of the program described in this thesis.
- Spike, or Action Potential (AP) - The electrochemical pulse that travels between neurons.
- Synapse - Biologically, this is the small gap between the dendrite of a sending cell and the axon of the receiving cell. In this model, the synapse represents the entire path between neurons.
- Thinking - The process of running NCS on a brain.

## 1.4 Necessary Biology

This section gives a brief review of some aspects of the biology of the brain that NCS is attempting to model. Those wishing more detail are referred to any good neurobiology text, such as [4].

Although the brain is composed of several parts, we are interested mainly in the cerebral cortex, the area in which it is believed that cognition takes place. Topologically, the cortex is a thin sheet on the outer surface of the brain. It is highly convoluted: folded in three dimensions so as to fit within the skull. It is composed of a large number of neurons (in the human brain, about $10^{11}$) and a similar number of glial cells of various types. Current thinking is that the glial cells play a secondary role in brain function, supporting and maintaining the neurons without being involved in cognition.

The cortex is divided vertically into layers that are distinguished by variations in the types of neurons. Most of the cortex has six of these layers; the hippocampus has only four. The cortex is also organized horizontally into columns, which are localized regions of high neural connectivity. It should be noted that columns and layers are not in any sense separate organs; they are distinguished anatomically only as patterns of cell distribution and connection. In NCS, they are used in the input constructs that define a particular brain and its organization, but otherwise have no function that can be modelled.

Each neuron is connected by axons and synapses to a number of other neurons (typically about a thousand). Neurons maintain a small electrical potential, normally about -70 mV, due to the interactions of channels in their cell walls with ions in the intercellular fluid. This voltage continually changes in response to external inputs, primarily stimuli received from incoming synapses. When the voltage reaches a critical threshold, these ion channels cause an abrupt rise and fall in the cell voltage. This is called an action potential (AP) or “firing a spike”.

This spike produces an electrochemical pulse which travels from the cell body, along outgoing axons and synapses, to destination cells. The propagation speed of this pulse is

relatively slow, under 10 m/sec, although it varies between different axon types. These pulses are nearly identical for cells of a particular type. Figure 1.1 shows the shape of a typical spike.

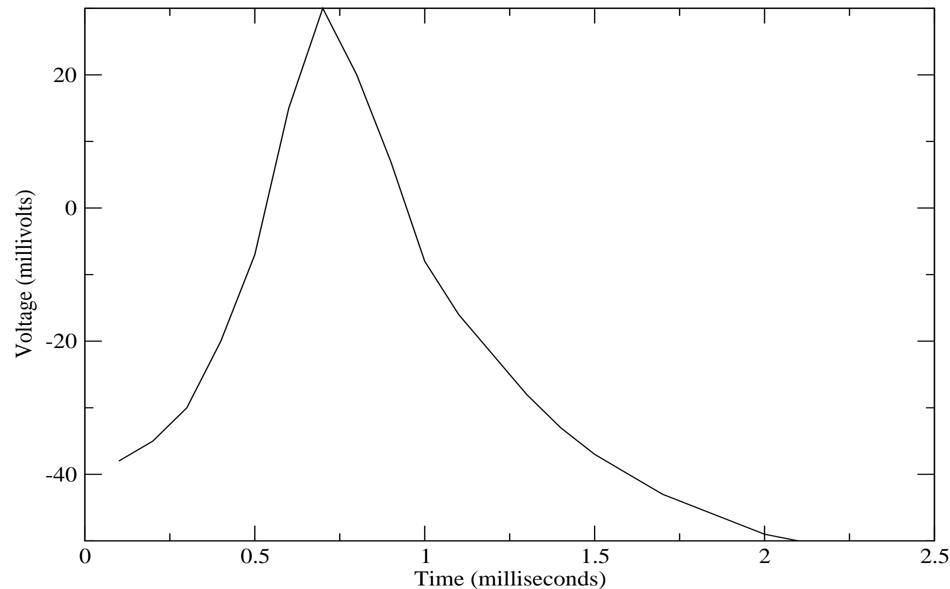


Figure 1.1: Typical Spike Profile.

When the spike pulse reaches the destination cell, it combines with the inputs from all the cell's other incoming synapses, which in turn may cause the destination cell to fire, thus continuing the process. Note that although anatomically the synapse is simply the gap between the axon of the transmitting nerve cell and the cell body of the receiving cell, in the program the entire mechanism that transmits this electrochemical pulse is contained in the `Synapse` object.

The behavior of some types of synapses changes in response to previous inputs, through a process known as Hebbian learning. A neuron's response to spikes may be increased (positive learning) or decreased (negative learning), depending on the number and timing of previous spikes.

## 1.5 Hardware

The NCS software currently runs on the Brain Computing Laboratory's dedicated Beowulf cluster, Cortex. Cortex presently consists of 64 compute nodes, each of which is a dual Pentium CPU motherboard with 4 GBytes of RAM. The cluster thus has a total of 128 processors, 60 Pentium IIIs at 1 GHz and 68 Pentium IVs [1] The software thus has available some 298,800 bogimips[2] of processing power.

| Packet Size (Bytes) | Packet Transfer Time ($\mu$Sec) | Transfer Rate (MBytes/Sec) |
|---|---|---|
| 128 | 25 | 4.9 |
| 256 | 25 | 9.8 |
| 512 | 25 | 19.5 |
| 1K | 25 | 39.1 |
| 2K | 50 | 39.1 |
| 4K | 50 | 78.1 |
| 8K | 101 | 77.5 |
| 16K | 126 | 124.0 |
| 32K | 201 | 155.5 |

Table 1.1: Myrinet/MPI Packet Times.

The nodes are interconnected *via* a high-speed Myrinet [12] network, providing a maximum transfer rate of 2.2 Gbits/sec. In practice there is considerable overhead imposed by the MPI library and other system-level software. Measured transfer rates for various packet sizes, using a standard `MPI_Send`/`MPI_Recv` protocol are given in table 1.1.

[1] The Intel Pentium 4 processor contains two virtual processors, which in theory should provide double the computing power. Their performance has proved disappointing in practice, however.

[2] The bogomips number is a performance measure readily available on any Linux system.

# Chapter 2

# Overview of Program Design and Operation

## 2.1 The Input File

NCS is intended for use primarily by a user community that is familiar with the anatomy of the brain, therefore the input format was designed to correspond to the structures found in the biological brain. A brain (at least in our present understanding) is organized hierarchically: it is composed of columns which are made up of layers which are made of different types of cells. Each cell contains compartments which are connected to the compartments of other cells by synapses. Compartments also contain substructures such as channels. The input file also allows for specification of inputs to the brain (*STIMULUS*) and outputs (*REPORT*).

## 2.2 Initialization

On startup, the program first collects some information about itself and the hardware on which it is running: the number of nodes and their compute power, process ids, and so on. Some of this information is written to a *kill file*[1] This file contains a line for each program sub-process with the node name, node number (MPI rank), and process ID of the program on that node. Collecting this information in one place allows ease of operations on all the

[1]The name reflects its original purpose, which was to provide a means of quickly killing all the processes of a misbehaving job.

processes: a misbehaving program may be killed, a debugger may be started, or the program resource usage may be monitored in a manner analogous to the Unix top utility.

## 2.3 Input and Parsing

After initialization, the program next must read and parse the input file(s)[2]. The parsing step is duplicated on each node, as the processing required is insignificant and the complex and error-prone code that would be needed to distribute the brain structures created by the parsing step is thereby avoided.

The NFS file server experiences contention problems when many nodes (more than 25 or so) simultaneously attempt to open and read from the same file. Therefore, input files are read only by the root node. Each file is read into a buffer, and MPI functions are then used to distribute this buffer to all of the other nodes. (Conceptually this distribution is an `MPI_Bcast`, but due to problems with the MPI broadcast of large blocks of data, it is implemented as a series of `MPI_Send` and `MPI_Recv` operations, each transmitting a fairly small packet.)

This buffer is then passed to the parsing module, which scans the input, checking for duplicate or undefined names and other errors. This module is a mini-compiler implemented in `YACC` and `Lex`[3]. The use of these tools allows the easy implementation of fairly sophisticated syntax checking and error reporting and makes it almost trivial to add features such as the use of variables and expressions in addition to simple numeric values.

If no errors are found in the input, the parsing module creates an `INPUT` structure containing the information in the input file in a form that is readily usable by subsequent code. Note that this structure contains the definitions from which the brain will be created. A particular definition may create many instances of the component it defines, or it may be present in the file but never used.

[2]The program receives one input file as an argument, but this may include further input files and other data files such as PSC templates or *STIMULUS* input. All are read and distributed in the same fashion.

[3]Actually the implementation uses the Gnu workalikes, `Bison` [5] and `Flex`[6]

## 2.4 Global Index Creation

The CellManager module now processes the parsed INPUT structure, and creates two index tables that allow subsequent code to locate the definitions of the objects it will be creating. These tables are the Global Cluster List (GCList) and the Connection Descriptor List (CDList). The GCList contains an entry for each cluster of cells defined in the input, while the CDList contains an entry for each connection between clusters. This entry contains pointers to the entries of the FROM and TO clusters in CDList and the synapse definition in INPUT. Once the CDList is created, the CellManager scans through it and determines the number of synapses for each connect[4]. The synapse count for each connection is added to the TO cluster's total for use by the distribution algorithm.

At this point, each node contains identical copies of the INPUT, GCList, and CDList structures.

## 2.5 Distribution

The components of the brain next must be distributed among the CPUs on which the program is running. The basic distribution unit is the cluster. Earlier code distributed individual cells, but this required excessive memory for the index tables. of memory because the global index needed to contain an entry for each cell. Distributing at the cluster level has the additional advantage that intra-cluster connections, which generally have the most stringent transmission time requirement, will always use the intra-node message-passing mechanism. (See section 4.3.)

Synapse processing dominates computation in any realistic brain, but because it is not possible to determine this load in advance, various heuristic distribution algorithms have been developed. The user may specify the choice of algorithm at run time. (The problem of designing an appropriate distribution algorithm will be discussed further in Section 4.1.)

[4]The specific cells that are connected will be determined randomly at a later point. Earlier code computed this information here and stored it, using an excessive amount of memory.

Once the distribution algorithm has computed cluster weights, the distribution routine processes the `GCList` and assigns each cluster of cells to a CPU.

As in the index creation step, at this point each node still contains identical information.

## 2.6 Brain Construction

Each node constructs an object of type `Brain`, which contains (among other things) an array of pointers to each of the `Cell` [5] objects created on the node. To construct this `Brain` each node scans through its copy of `GCList` and, for each cluster assigned to the node, creates the specified number of `Cell` objects and all the components contained within the particular cell type.

Each `Cell` likewise contains an array of pointers to the `Cell`'s `Compartment` objects. Conceptually, each compartment thus has an internal address which is a triple of numbers: (Node, Cell, Compartment), which the `MessageBus` uses to pass between the compartments of the brain information such as stimulus inputs, report output, and most importantly synapse firings. For efficiency, this NCC address scheme has been superseded by one that uses the physical memory address of the receiving object. See Section 15.

When this step is completed, the structures on each of the nodes will differ, with each node containing some fraction of the cells that make up the complete brain.

## 2.7 Connection

Now that each node has been populated with its share of cells, the program must establish the connections between them. Recall from Section 1.4 that these connections, or synapses, are one-way communication channels along which an action potential propagates. Each connection thus involves two compartments, the *From* or sending compartment and the *To* or receiving compartment. The connection exists in two parts: on the *From* side, each cell must

[5] In the current implementation, a Cell object is simply a shell that serves as a container for the compartments from which the cell is constructed. Computation takes place in the compartment and the elements contained within it.

know where to send messages when it fires an AP. On the *To* side the cell not know where the AP originated, but must contain the necessary code to process the messages as they arrive.

The connection process thus consists of each node determining which of the cells that reside on it are to be connected, finding the node on which the other end of the connection resides, and exchanging the information needed to create the necessary data structures on both sides. Implementing an efficient solution to this problem is not a trivial process. It will be discussed in Section 4.2.

At this point in the program, the brain has essentially been created.

## 2.8 Stimulus and Report Creation

The brain now needs something to think about and some way to communicate its "thoughts" to the outside world. This is the function of the `Stimulus` and `Report` objects, respectively. These objects are essentially mirror images. They are both based on the same *(CLCC)* paradigm used in the `CONNECT`. Each `Stimulus` object delivers input messages to the specified *CLCC* group, and each `Report` object retrieves information from a specified *CLCC* group and writes it to output. The input and output channels are usually files, but the program has the capability to read and write sockets as well, so that it may interact with the world in real time [9, 10].

The same `GCList` created by the connection manager is used to determine which nodes contain the *CLCC* group for each `Stimulus` or `Report`. If a node does contain the cells, an object is created and placed in the `Brain`'s list.

## 2.9 Thinking

The initialization process is now complete, and the program is ready to begin "thinking"; *i.e.* processing input stimuli. At the highest level this is a simple loop (in `Brain::DoThink`) which iterates over the number of timesteps specified in the input. At each iteration, each node performs the following steps:

- Process the stimulus objects for the node. These objects create stimulus messages that are dispatched to cells on the same node. Unlike synapse messages, these messages must be delivered at the same timestep in which they are created.

- Call the `MessageBus` function to deliver to the destination compartments messages created by stimulii or received from other nodes.

- Loop through the list of cells on the node, calling each one's `DoProcessCell` function to invoke its computation. This in turn calls each of the cell's `DoProcessCompartment` function.

- Call the `MessageBus` function to ensure that all message packets created at this timestep have been started on their way to the destination nodes.

- Process the reports

- Ensure synchronization. For performance (as discussed in Section 4.3), this is not a simple barrier at the end of the timestep, but it can be approximated as such.

Once the the specified number of timesteps has been completed, the program does any required cleanup work and then terminates.

## 2.10 Internal Cell Processing

The simple statement "process the cell" hides the bulk of the computation of cellular and synaptic dynamics that are being simulated. The remainder of this chapter expands on that statement.

As mentioned previously, the `Cell` object is merely a container for one or more `Compartment`s, so that the `DoProcessCell` function consists simply of calling each `Compartment` object's `DoProcessCompartment` function.

## 2.11 Compartment Processing

A compartment has an internal state which consists of various biologically-derived parameters. Additionally, a compartment will contain some or all of the following types of objects:.

- `Channel` objects of various types. These objects simulate various components of the biological neuron, and contribute the channel current $I_{chan}$ to the compartment.
- `SendTo` structures, which specify to which cells the firing messages are to be sent. These are derived from the From side in the connection step.
- `Synapse` objects, derived from the To side in the connection step. These objects simulate inputs received from other cells, and contribute the synapse current $I_{syn}$ to the compartment.

There may be an arbitrary number of objects of each type, implemented as arrays or lists.

The compartment state is reflected in the membrane voltage $V_m$. In a quiescent cell, this has a resting value $V_{rest}$. The activities of input stimulii, channels, and synapses all create currents which drive $V_m$ away from $V_{rest}$, while the leakage current $I_{Leak} = G_{Leak}(V_m - V_{rest})$ acts to return $V_m$ back to $V_{rest}$. Thus a compartment's membrane voltage normally is determined by the equation

$$V_m[i] = V_{rest} + (V_m[i-1] - V_{rest})P + \frac{\Delta t}{C} I_{total}$$

where

- $P$ is the compartment's persistence;
- $C$ is the compartment's capacitance;
- $\Delta t$ is the length of a timestep, in seconds; and
- $I_{total} = I_{stim} + I_{chan} + I_{syn} - I_{Leak}$.

However, the real-world behavior of the compartment voltage is highly non-linear: when $V_m$ reaches a particular threshold value, it increases rapidly ("spikes"), the cell fires an action potential, and the voltage quickly collapses towards the compartment's resting potential $V_{rest}$. This spike behavior is essentially identical for all cells of a given type and is modelled as a spike template (Figure 1.1), rather than being explicitly computed.

In general, the processing loop for a compartment does the following:

- Process all the incoming messages for the timestep. These may be either stimulus or spike (synapse firing) messages. Stimulus messages simply modify the compartment's internal voltage or current. Spike messages place the corresponding synapse on the active list (if it is not already active) or add a new Post-Synaptic Conductance (PSC) waveform to it if it is already active.

- Compute the channel current. This current modifies the internal state of the compartment. If the threshold voltage is reached, the compartment fires. The `SendTo` list is used to generate spike messages for all destination cells.

- Process the active synapses and channels. Their outputs modify the internal state of the compartment. If the threshold voltage is reached, the compartment fires. The `SendTo` list is used to generate spike messages for all destination cells.

## 2.12 Channels

There are several types (or *families* in the input) of `Channel` objects. Each family is quite simple and merely computes the channel current as an exponential function of the compartment voltage, the specific equations for each type being determined from experimental data. The owning compartment simply sums the current contributions of all its channels.

## 2.13 Synapse and Spike Processing

`Synapse` objects interact with the owning compartment in a more complicated fashion. A `Synapse` is *active* only if it has recently[6] received a spike message. On receipt of a message, it updates its USE[7] and RSE[8] values according to the general equations

$$
\begin{aligned}
USE[i] &= USE_{base} + (1 - USE_{base})(USE[i-1]e^{\frac{-\Delta t}{\tau_{Facil}}}) \\
RSE[i] &= 1 + (RSE[i-1](1 - USE) - 1)e^{\frac{-\Delta t}{\tau_{Depr}}})
\end{aligned}
$$

and uses them to compute the conductance value $G_{syn}$ for the synapse, as $USE * RSE * G_{Max}$. Note that any particular synapse type may be specified to compute either USE or RSE, both, or neither (in which case the $G_{syn}$ is simply $USE_{base} * G_{Max}$).

$G_{syn}$ is then used in conjunction with the PSC waveform template to compute the contribution of that spike to the compartment's synapse current. The `Synapse` places an `ActiveSynPtr` structure on its compartment's `ActiveList`. The structure contains the conductance value, a pointer back to the synapse object, and an index into the synapse's PSC template array. At each timestep, the compartment code steps to the next template value and computes that synapse's contribution to the synapse current as $I = G_{syn} * PSG[i](V_{SR} - V_m)$, where $V_{SR}$ is a property of the synapse called *Synapse Reversal*, which is fixed for any particular synapse. The `ActiveSyn` structure's index is decremented and, if it is zero (meaning it has reached the end of its template), it is removed from the list.

At each timestep, the compartment sums the current contributions of all the active synapses on the `ActiveList`. Because at any time there may be hundreds or thousands of these, the processing of these lists is a major factor in performance. (See Section 17 for measurements.) Consequently, two potential optimizations present themselves.

The first optimization entails simply reducing the length of the PSC template. As shown in Figure 2.1, the typical PSC waveform rises quickly to a maximum and then decays expo-

[6] That is, within a number of timesteps determined by the length of its PSC template.
[7] Utilization of Synaptic Efficacy
[8] Reduction of Synaptic Efficacy

nentially, so that the tail of the waveform contributes relatively little to the synapse current and might be neglected with little effect. Preliminary tests show that this is indeed the case: shortening the template produces a speedup roughly proportional to the amount of shortening. The PSC templates are read from files, however, so this is a decision that can and should be made by the brain designer rather than the application programmer.

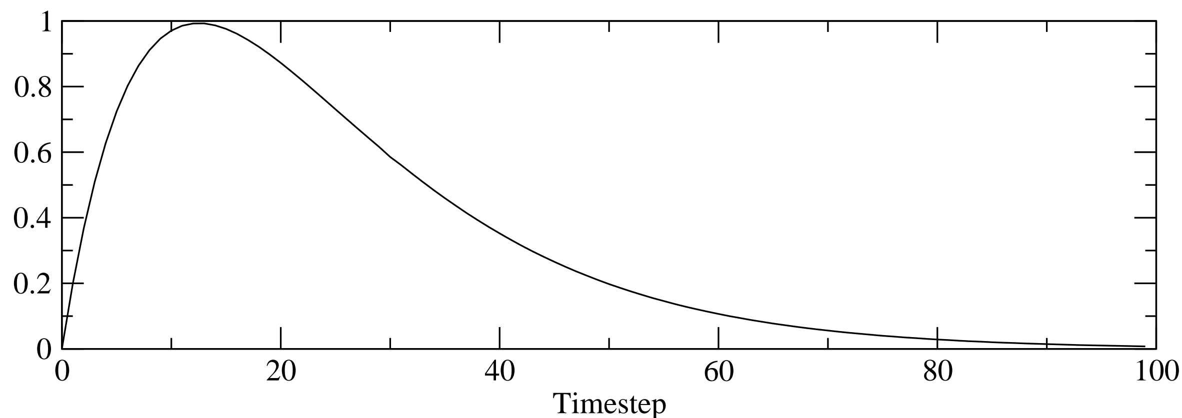


Figure 2.1: Typical Post-Synaptic Conductance (PSC) Template.

The second optimization can be done if all synapses use the same PSC template (which currently is the case in practice). A compartment will frequently receive more than one synapse input at any given timestep. The summation of these synapses can be done at the receiving timestep, with one combined `ActiveSynPtr` added to the list, rather than many.

## 2.14 Synapse Learning

Synapses also may exhibit a property known as *Hebbian learning*, in which the response of the synapse is modified, depending on how recently it had received previous spikes (positive learning), or how recently the owning compartment itself had fired (negative learning).

Learning is implemented *via* learning tables. For negative learning, when the synapse receives an incoming spike message, it computes the number of timesteps since the previous message and, (if the value is less than the table length, looks up the corresponding value in the negative learning table and subtracts it from the synapse's current USE value. This has

the effect of reducing the current contributed to the compartment by the incoming spike.

Positive learning is similar but is triggered by the firing of a compartment. The compartment scans through all its synapses that have positive learning enabled and, for each, computes the time since that synapse last received a spike, looks up the corresponding value in the positive learning table, and increases the synapses USE value by that amount.

# Chapter 3

# Profiling Tools

In attempting to model biologically interesting properties of the brain within a realistic time frame, the NCS program is pushing the limits of current technology in both processor speed and memory usage. Considerable effort has therefore been devoted to optimizing the code for both speed and size.

Many compute-intensive programs consist of a few loops iterating many times over the same code, so changes in performance can be adequately measured with relatively coarse-grained tools. An NCS brain, however, is typically composed of a large number of different elements which are processed in no particular order. Therefore, unless timing measurements can be made to a precision on the order of tens or hundreds of machine cycles, the effect of a change to some element may be lost, even though the change might be highly significant to the performance of that element.

Currently there do not appear to be any tools, especially for parallel programs, that allow such precise timing measurements and which are both useable and readily available. Thus in order to efficiently do the optimization work described in this thesis, it was necessary to develop a set of tools which could accurately measure both elapsed time (or number of machine instructions executed) and memory usage.

This chapter describes those tools and their use. They are of two sorts: functions called from within the program, which collect information for later analysis, and programs that use external information to monitor run-time progress and resource statistics.

## 3.1 Internal Profiling Routines

The internal profiling routines that were developed fall into two categories: those that profile execution speed, and those that profile memory use. Although the methods used in the two categories are different, the functions and their outputs have been combined in a single package for convenience.

### 3.1.1 Profiling Execution Time

There are two basic methods of profiling code. The first is to pre-process the code, inserting data collection statements in the executable at regular intervals. The output is then analyzed, and the fraction of time spent in any particular section can be calculated. This is the approach used by e.g. the classic `prof/gprof` interface.

Unfortunately, executing the data collection statements typically will increase a program's execution time by an order of magnitude or more. While this overhead is not itself counted in the profiling (so that results remain valid), the increased run time may, and most certainly does in the case of NCS, make profiling runs impractical. Furthermore, parallel programs are often dependent on the relative timing of events in different execution threads, so that the increase in execution time changes the behavior of the program. Probably for these reasons, there do not seem to be any attempts at parallel profiling packages using this approach.

The second method is simply to measure the execution time of particular segments of code, using some external or internal clock as a reference. This is more practical in that it does not significantly change the executing code or its timing. The difficulty lies in finding a clock that is both accurate and fine-grained enough to measure short sections of code. Readily available sources, such as the Linux `time` and MPI `MPI_Wtime` functions, have resolutions in the microsecond range at best, while CPUs currently operate at clock frequencies of 1 GHz and above. Thus during each tick of these clocks, several thousand machine instructions may be executed.

To address this problem, many current microprocessors implement an internal hardware clock of some sort. In the Intel Pentium family, the clock is a 128-bit counter that is incremented at each processor cycle. The counter is accessible from user code by means of the `RDTSC` assembly language instruction.

There are a number of technicalities, mostly related to instruction scheduling and pipelining, that limit the practical resolution of this clock to several tens of processor cycles. These are addressed in [3]. However, it is still far more accurate than any other time source. For the purposes of this code, it is sufficient to know that the RDTSC instruction is atomic, that the value is returned as a 128-bit `unsigned long long` that is the number of processor cycles since the processor was rebooted, and that the rollover period of the counter is on the order of 100 years.

Presently there are several profiling packages described in the literature[13, 15] that use this approach. The available ones were tested, but none proved usable with the NCS code, being tied to particular environments and/or C++ dialects or requiring the installation of kernel patches that might have conflicted with the cluster management software. These packages also seemed rather over-ambitious, attempting (but failing) to automate the instrumentation of the code or providing GUI analysis tools which did not display useful output. For these reasons, it was necessary to develop a simple profiling package for use with NCS.

## 3.2 The QQ Profiling Library

The profiling package is called QQ. It is by no means tied to NCS: it should be possible to use it as-is with any program, parallel or sequential, that runs on an Intel Pentium based Linux platform and is compiled with a gcc-based compiler. It should also be easy to adapt it to other processor families with similar internal counters by replacing the assembly language `RDTSC` instruction with one appropriate to the hardware.

### 3.2.1 Basic Theory of Operation

The principal development goal of the QQ timing code was to interfere as little as possible with existing code. The profiling operations themselves should execute as quickly as prossible, so as to cause minimal change to program timing and increase in execution time. It was likewise desireable to be able to insert the needed instrumentation calls in the code in a manner that would distract as little as possible from code readability, and which could be turned on and off with a simple compile flag.

QQ is based on the notion of recording named events. There are several types of events, each with some some associated information. All have in common a key, which identifies the particular event, and an event time. Depending on the type, there may be other information such as a value, count, or state flag. The various event types are combined into an event union.

When the profiler is initialized by calling the QQInit function, it allocates memory for some specified number of events, initializes the event pointer to the first event, and sets the base time to the current `RDTSC` value.

After initialization, one or more of the `QQAdd*` functions are called to add event types to the internal name table. Each function is passed a name (character string) for the type of event, and returns the integer key for the event. These keys are variables in program space, and are the only parts of the QQ timing package that exist in code when it is compiled with the `QQ_ENABLE` flag off.

The individual event recording functions are thus reduced to a minimum: each checks to see if the event pointer has exhausted the allocated number of events. If not, the `RDTSC` instruction is called, the returned value, the key, and any other information is written to the current event, and the event counter is incremented.

The `QQRecord` function allows event recording to be turned off and on under program control. It does this by caching the current event pointer and replacing it with a value greater than the maximmum allocated, thus allowing the event recording functions to use a single

test to determine whether or not an event is to be recorded.

When profiling has finished, the `QQOut` function is called to write the saved event information to a file. For parallel programs, the output from all nodes is combined into a single file, along with information to identify each node. This file can later be read by a simple C program, and the information converted to the forms needed by various analysis tools.

Figure 3.1 shows an example of such a tool: several timesteps of an NCS run are shown, with the times spent in various sections of the code.

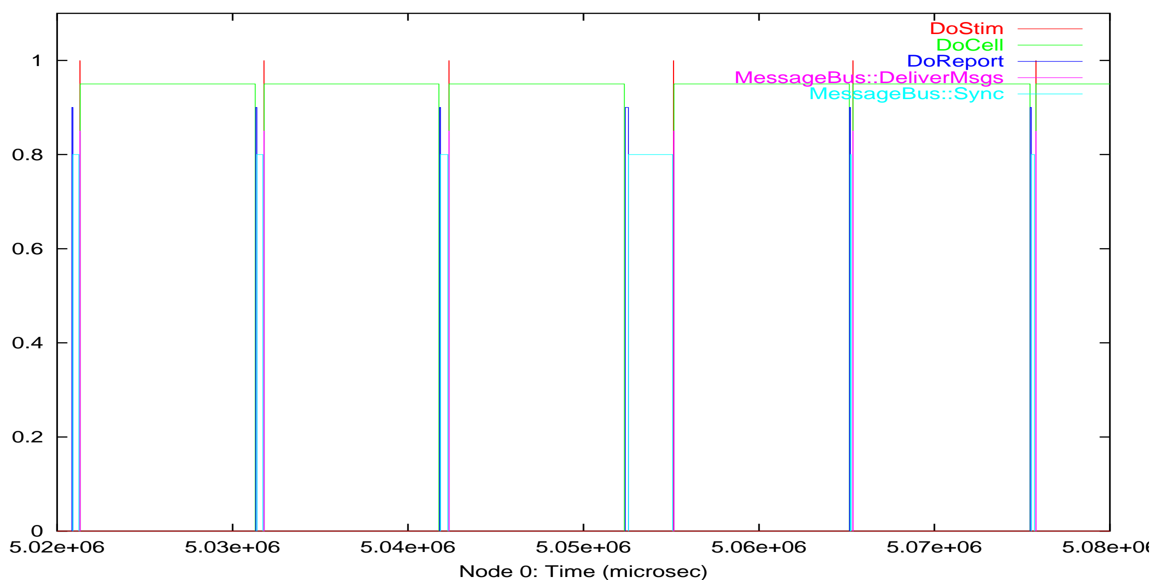


Figure 3.1: Plot of Typical QQ Output.

## 3.3 Profiling Memory Use

Profiling a program's memory usage is a more difficult task than profiling execution time. Indeed, it does not appear that a completely satisfactory solution is possible. However, is is possible to gather much useful information, and the routines to do so are described here.

First, in a Linux system it is possible to obtain the total memory allocated to a process at any point in time by reading the `statm` pseudo-file in the process directory of the `/proc` filesystem. This file contains the number of 4 KByte pages allocated to the process. The

number includes both code and data space, and may include memory owned but not actually in use. It is thus is an upper bound on the memory used by the program at any point.

The `GetMemoryUsed` function uses this method to return the total memory allocated to the program at the time when the function is called. By careful use of this function, it is possible to obtain a good idea of how much memory is used by sections of code. However, it does not allow for measurements on the scale of individual structures or objects.

The system allocates memory to a program in units of pages. It is up to the internal memory allocator (generally in the `malloc` library) to parcel the pages into smaller units. If this memory allocation is done directly by calls to `malloc` library functions, it is easily measured by using the preprocessor to redefine the function call to a different function, which records allocation information and passes the operation through to the actual allocation function.

Thus, for example, the `malloc` function can be redefined to be

```
#ifdef MEM_STATS
    #define malloc(arg) MemMalloc (MEM_KEY, arg)
#endif
```

Each call to malloc in the code now becomes a call to the `MemMalloc` function, and the new call contains an additional argument, which is the key under which the allocated memory will be recorded. Each chunk of allocated memory is stored in a C++ `map` object, indexed by its address (the pointer value returned by the alloc call) The call to `free` is redefined to remove the item corresponding to the address from the map. In this way, a consistent record of all currently-allocated memory is maintained, and the record for each item may contain information on the type of object, what routine allocated it, and so forth.

### 3.3.1 Object Creation

In principle it should be possible to do something similar to the above for objects by overloading the new and delete operators. This has not yet been done in practice. Instead, a simpler method was used. This method requires more work from the programmer, but gives useful results.

Eeach object constructor must have calls to the `MEMADDOBJECT` and `MEMFREEOBJECT` routines added to them. When memory profiling is turned off, these calls evaluate to empty statements; when it is on, they evaluate to calls to the memory recording functions and contain the object's this pointer and the sizeof (this) value as arguments.

These functions allow most explicitly allocated memory and objects defined in the code, to be recorded. Memory that is allocated internally by various library functions or standard C++ objects can not be recorded, however. The information available, although not complete, is still quite useful. If nothing else, the amount of recorded as allocated by the profiler can be compared to the total memory allocated to the program. If the two quantities are significantly different, the memory is probably being allocated somewhere internally.

## 3.4 External Monitoring Programs

In Linux, the `/proc` filesystem contains a good deal of information about the state of the system and of individual processes, The problem is one of extraction and organization. At this writing, the Cortex cluster is composed of 64 separate Linux operating systems, each controlling two CPUs. Parallel programs such as NCS may run on any or all of these nodes. Indeed, a node may have several program processes running on it, for instance to take advantage of the P4 chip's second (virtual) CPU. How then does one identify the subset of available information that is relevent to the program one is examining?

One solution to this was addressed briefly in section 2.2 On startup, each process in the parallel program extracts identifying information from the system: the name of the machine within the cluster, its process ID, and its node number or MPI rank. That information is

gathered to the root node via MPI and is written to a ”kill file”. Programs can use this information to extract almost any item of information about a program that is available from the system and organize it in a useful form. The variety of possible programs is unlimited, but two useful examples will be briefly discussed here.

### 3.4.1 memsnoop

The memsnoop program is somewhat analogous to the *nix top utility. Large NCS models tend to push the hardware limits of even the Cortex cluster, and so programs will fail for reasons such as memory exhaustion. The MPI system often does not return an identifiable error when this happens, because the other processes eventually block on communications with the dead node.

The memsnoop program provides a means of monitoring process status and available memory on all the nodes of a process. This consists of a simpe perl script which read the killfile information, and uses the ssh remote execution facility to repeatedly execute the command

```
cat /proc/meminfo | grep MemFree
```

on each node. The returned line was scanned for the amount of free memory remaining, the amounts sorted, and the several lowest displayed, thus identifying the problem node.

### 3.4.2 Parallel gdb

The use of a debugger such as gdb[7] can be considered to be an extreme case of performance monitoring. Although gdb is not equipped with an intentional parallel debugging ability, it does have features which, with the application of some ingenuity, will allow some simple but extremely useful, parallel debugging to be done. This is most useful in cases where one or more processes encounter an unexplained segfault, or where the section of code that needs debugging can be located fairly precisely.

`gdb` has the ability to attach itself to a running process. Since the killfile contains the process ID and machine name of each process, a script can be written that invokes an instance of `gdb` which attaches to each process. (The executable being debugged must have some delay inserted after the initialization/killfile creation step in order to allow the user to run the parallel `gdb` script.) The script opens an `xterm` window for each instance, and debugging interaction can be done in each window.

`gdb` also has the ability to read commands from a script file.

By default, `gdb` simply resumes running the executable it has attached to. Each process thus continues until the program terminates, either normally or with an error. At that point, control returns to the user, and standard `gdb` commands such as `bt` and `frame` can be used in the process window to localize the fault and examine code and data in order to determine the cause.

Other `gdb` commands can be specified in a user-defined command file, so the user may for instance specify particular breakpoints. Every process will continue until it reaches a breakpoint, and then return control to the user. Different nodes may of course follow different paths through the code, so the interaction in each process window may be different.

# Chapter 4

# Improving Parallel Performance

Much of the work of optimizing the NCS code has been essentially sequential programming: after all, parallelizing an inefficient sequential program yields an inefficient parallel program. This sequential optimization has been mostly the repeated application of tools discussed in section 3.1 to measure performance, locate speed bottlenecks or excess memory use, improve that section of code, and repeat the process. While valuable, this type of work is not the stuff of which a thesis is made.

Three factors limit the performance of NCS as a parallel program: load imbalance, message-passing overhead, and synchronization. Each of these areas was addressed in this work, and this chapter describes the solutions that were developed.

## 4.1 Load Balancing

Load balancing is the process of attempting to distribute work evenly amongst the CPUs on which a parallel program runs, so that all processors are fully utilized. For many applications this is a simple matter of assigning an equal number of work units to each node. The NCS3 program used this method, but for a number of reasons it proved less than satisfactory in practice:

- The Cortex cluster originally was composed of 60 1.0 GHz CPUs. It was later expanded to 128 CPUs, with the added CPUs the having about 3 times the compute power of the originals. This difference must be factored into any load balancing scheme.

- Cells are not identical: they are constructed from a number of different components such as compartments, channels, and so on. Each cell type therefore has a different computational weight, which can be approximated as the sum of the components.

- In all but the most trivial models, the number of synapses exceeds the number of cells by a large factor. (Recall from Section 1.4 that a typical cell may have more than a thousand synapses.) Thus synapse processing is generally the most significant component of computational load.

- When running models on the order of $10^9$ synapses, it is necessary to balance memory use rather than computation. Each synapse requires 56 bytes, plus 16 bytes for each active spike[1]. The number of active spikes at any instant varies according to input but is typically on the same order as the number of synapses. The Cortex cluster has only 256 Gbytes of RAM, so some imbalance in computation must be accepted in order to allow large models to fit in physical memory.

Factoring synapses into the load-balancing process is complicated by the fact that computation takes place on a particular synapse only when the synapse is in the firing state. It is not possible to predict when or how often a synapse fires because firing is determined by the input stimuli. Indeed, obtaining the patterns of synapse firing is the goal of an NCS simulation.

This unpredictability applies to memory usage as well: the amount of memory needed to construct the brain can be computed as the sum of its components, but a running brain needs significant additional memory to hold the dynamic information that represents synapse firing states. The exact amount required is impossible to predict, but in practice the current implementation seems to require about equal amounts of memory for static and dynamic data.

[1]As of SVN revision 14 of the code, compiled without the `SAME_PSC` option. With the `SAME_PSC` option, usage is somewhat less but even more difficult to determine exactly. See Section 2.13 for a detailed explanation.

In order to include synapse weights in the load balancing calculations, the number of connections between each pair of cell clusters must be known before distribution. This, in turn, implies some sort of global address table or lookup mechanism from which each node may obtain the necessary information for the other ends of the synapses its cells own.

Because each node potentially needs information on any cell, the simplest way to implement this lookup mechanism is to create an identical copy of the table on each node. Early versions used a lookup table with entries for each cell, which was adequate for fairly small models. As the number of synapses approached 50 million, however, the amount of memory used to store this table was approximately 1 gigabyte. Obviously, a more compact table was needed.

Currently the table is based on cell clusters. (Recall from Section 1.3 that a cluster is a group of cells defined by the `CELL_TYPE` statement.) This table configuration reduces the memory requirement to a few megabytes, rather than gigabytes, with the reduction factor depending on the number of cells per cluster. It is also convenient in other ways, because the input file format describes inputs, outputs, and connections between cells in terms of clusters.

### 4.1.1 Load Balancing Algorithm

Any balancing algorithm must have some way to determine the compute power of the processors on which the program is running and the amount of computation to be distributed among those processors. Linux provides a simple and reasonably accurate performance measure in the bogomips number. This number is calculated as part of the operating system boot process and is always available in the `/proc/cpuinfo` pseudofile. In the initialization phase of an NCS process, each node reads its bogomips number, and these individual values are gathered (with `MPI_Allgather`) onto every node. The sum of these values is the total compute power available to the program, and thus the share contributed by each processor is a simple fraction. The total amount of computation is determined by assigning some weight, say 1.0 for simplicity, to a basic cell and ratios of this amount to other components that may

be added. Table 4.1 shows performance numbers for some typical components. This simple weighting scheme is complicated, however, because any particular synapse will fire, and thus be active in computation, during only a small fraction of timesteps, and the firing of any particular synapse or group of synapses is unpredictable. In practice a heuristic factor must be applied to the synapse weights. After this factor is applied, a simple summation gives a total compute weight for each cluster. Then, using the cluster weights, clusters can be assigned to each node according to some scheme which balances the compute load according to the computing power available on each processor.

| Item | Time ($\mu$Sec) | Load Factor |
|---|---|---|
| Base Cell | 0.246 | 1.0 |
| Kahp Channel | 1.541 | 6.2 |
| Km Channel | 0.152 | 8.4 |
| Ka Channel | 0.152 | 9.8 |
| Synapse[a] | 0.305 | 1.2 |
| Synapse[b] | 0.855 | 3.4 |
| Synapse[c] | 0.927 | 3.7 |

a) RSE = NONE, Learn = NONE
b) RSE = BOTH, Learn = NONE
c) RSE = BOTH, Learn = BOTH

Table 4.1: Compute Factors: Time required to process one instance of each element, as measured on a 700 MHz Intel Pentium III processor.

The simplest scheme is to assign clusters to nodes in a round-robin fashion, so that each node receives clusters in turn until it either exceeds its assigned weight or all clusters are assigned. More complicated schemes are possible. For instance, cells within the same cell grouping (cluster, layer, column) tend to communicate more frequently within the group than across groups. Because message passing between cells on the same node is far cheaper than between cells on different nodes, an effective load-balancing algorithm might try to assign clusters to nodes in such a way as to minimize inter-node communication.

### 4.1.2 Memory Balancing

As previously mentioned, in order to run the largest models it is necessary to balance memory use, rather than compute load, and accept the resulting inefficiency. The user selects this option by setting the input file's *DISTRIBUTE* variable[2] to the value "BySynapse", telling the code to distribute an equal amount of memory to each node. In theory, the memory sizes of different brain components could be used in this distribution. In practice, the memory used by synapses usually exceeds that used by the rest of the brain by at least two orders of magnitude. In the current implementation all synapses are the same size, so memory balancing simply counts synapses.

Tables 4.2 and 4.3 show the results of balancing the same brain for computation and memory use. Nodes 2 and 3 have P3 processors; nodes 0 and 1 have P4's, with about 2.5 times the processing power.

| Node | Clusters Assigned | Cells Assigned | Synapses Assigned | Bogomips of Node | Weight Assigned |
|---|---|---|---|---|---|
| 0 | 411 | 4862 | 239912 | 4347.9 | 28825.4 |
| 1 | 396 | 4680 | 239704 | 4331.7 | 28650.4 |
| 2 | 174 | 2060 | 110070 | 1972.0 | 13067.0 |
| 3 | 173 | 2048 | 109934 | 1972.0 | 13041.4 |

Table 4.2: Example of Load Balancing.

| Node | Clusters Assigned | Cells Assigned | Synapses Assigned | Bogomips of Node | Weight Assigned |
|---|---|---|---|---|---|
| 0 | 298 | 3540 | 180789 | 4347.9 | 9770692.0 |
| 1 | 310 | 3666 | 173164 | 4331.7 | 9811472.0 |
| 2 | 273 | 3220 | 172839 | 1972.0 | 9765588.0 |
| 3 | 273 | 3224 | 172828 | 1972.0 | 9772880.0 |

Table 4.3: Example of Memory Balancing.

[2] The code is designed to allow any of a number of distribution schemes to be selected by specifying different values for *DISTRIBUTE*.

### 4.1.3 Implementation

As described above, the load distribution and connection process has four steps:

1. Create the cluster and connection information from the input, counting the number of cells and connections.
2. Assign weights to the clusters according to load factors (Table 4.1) or memory usage (Table 5.4).
3. Assign clusters to nodes according to some algorithm that attempts to optimize computational load, memory use, and communication.
4. Create the cells and connections.

## 4.2 Connections

Once clusters have been distributed to nodes and the cell objects created, the connections between them (that is, the synapses) must be determined.

### 4.2.1 Creating Connections

When two clusters of cells are to be connected, the connection is seldom all-to-all (that is, connecting every cell in one cluster to every cell in the other) because this is both biologically unrealistic and computationally infeasible. Instead, the brain designer specifies a connection probability, and this fraction of the possible connections, chosen at random, are created. More precisely, given two clusters *FROM* and *TO*, with $M$ and $N$ cells respectively, and a connection probability $P$<=1.0, a connect specifier will create $M \times N \times P$ synapses. Note that it is also a requirement that the connections created by any specifier be unique: for any $i$ and $j$, there should be at most one synapse *FROM* [$i$] -> *TO* [$j$].

Previous versions of NCS used a simple probability test to determine connections: loop through both $M$ and $N$, generate a random number each time and, if that number is less than

the specified connection probability, make the connection. While this approach is adequate for small numbers of cells, it is obviously $O(n^2)$ with respect to the product[3] of $M \times N$. This factor was, in large part, responsible for the excessive setup times for larger brain models.

A simple algorithm that is $O(n^2)$ with respect to the number of connections created is possible. Because the connection probability is generally quite small, this approach yields much shorter startup times. The algorithm requires an $M \times N$ connection map array. Two random numbers are selected in the ranges $[0...M-1]$ and $[0...N-1]$. This pair indexes an entry in the map array. If the entry is not set, it becomes set, and the total number of connections made is incremented. If it is set, a new pair of random indices is generated. In either case, the process continues until the required number of pairs are generated. (For probabilities $> 0.5$, the obvious inversion is used: all connections are initially assumed to exist, and the random process deletes them until the desired number is reached.) Because connection probabilities are generally on the order of 0.1 or less, duplicates are quite rare, and the algorithm is essentially $O(n)$ with respect to the number of connections made.

Figure 4.1 shows the time taken to determine the connections between two clusters as a function of cluster size. The clusters are the same size, and the connection probability is 0.1. (Note that with a cluster size of 10,000 NCS3 fails due to memory exhaustion.)

### 4.2.2 Making Connections

Recall that each connection, or synapse, is a one-way communication channel from some cell to some other cell. (In actuality, from a compartment in the *FROM* cell to a compartment in the *TO* cell.) A cell sends firing messages out on all synapses for which it is the *FROM* end, and so it must know where to send the messages. This is implemented as the compartment's `SendTo` list: an array containing the *(Node, Cell, Compartment, Synapse)* address of each of the compartment's *TO* compartments.

On the *TO* side, each cell likewise maintains an array of `Synapse` objects, each of which

[3] This refers to the determination of the connections only. The actual synapse creation is done later, and is proportional to the number of connections made.

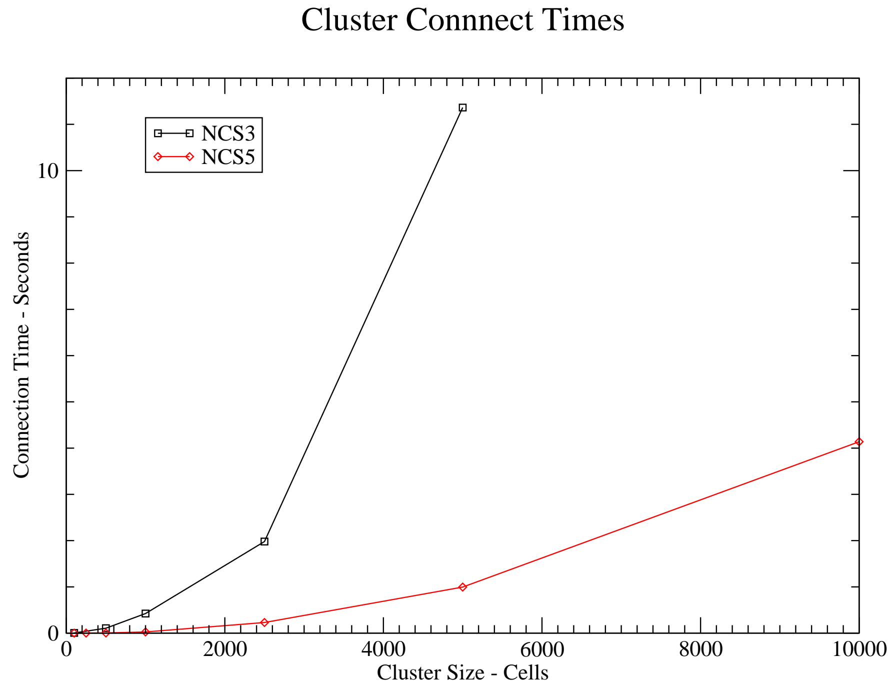


Figure 4.1: Connection Times.

is the destination of a `SendTo`. This object is where the actual computation for the synapse takes place.

At first sight, there seems to be a circularity here. The cells must be assigned to nodes before the connections can be made, but the connections (or at least their number) must be known to do the load balancing and distribution. By appropriate design of the connection algorithm, however, the information in each *CONNECT* statement serves to specify the exact number of synapses that are to be created, even though the specific cells to be connected by them will not be determined until later.

The connection algorithm thus proceeds as follows:

1. On every node, loop through the connect descriptor list.

2. If the current node is the *TO* side of the connect, determine which particular cells are to be connected. This is done with a connect map: pairs of random numbers are selected to determine the *FROM* and *TO* cells, and the map is marked to prevent duplicates. For

small connection probability (which is nearly always the case in practice), this scheme is nearly $O(n)$.

3. If the *FROM* side of the connect is on this node as well, process the information directly; otherwise use MPI to transmit a copy to the *FROM* node.

4. If the current node is the *FROM* side, but not also the *TO* side, it waits for the information to be received from MPI. Sends and receives will match because each node processes the list of *FROM-TO* pairs in the same order, eliminating any possibility of deadlock.

5. The *FROM* side uses the information to create a `SendTo` object for the specified compartment and stores a pointer to it in a temporary vector.

6. The *TO* side likewise uses the information to create a `Synapse` object for the destination compartment and stores a pointer to it in a temporary vector.

7. When all connects have been processed, loop through all the compartments on the node, allocate permanent arrays for both `Synapse`s and `SendTo`s, and copy the pointers to them. For efficiency, each compartment's lists can be sorted.

## 4.3 Message Passing & Synchronization

Although, as will be explained in Section 5.2.1, it is not possible to make direct comparisons of the execution speed between NCS3 and NCS5, an examination of the surviving versions of the code and documentation in [18] make it clear that the message passing scheme used by NCS3 most likely had a number of inefficiencies. The most notable of these was the use of the same communicator and message format for distributing stimulus and report data and the synapse firing messages. This required the inclusion of a message type field in the message packet, as well as additional overhead needed to distribute messages of different kinds to the proper destinations.

In addition, messages were pre-allocated, with a 60-byte message object allocated for every synapse. This wasted memory, because only a small fraction of synapses (typically less than 1%) are actively firing (and thus transmitting a message) during any particular timestep.

NCS5 separates the three functions. Stimulus messages and reports are now produced locally on each node[4]. This approach reduces the traffic on the network and, along with other optimizations, allows the size of the individual synapse firing message to be reduced from 60 to 20 bytes. In the NCS3 message packaging scheme, each message transmitted about 40 bytes of unnecessary information, resulting in a 200% overhead.

While these changes improved performance significantly, further analysis showed that more improvement was possible. The old algorithm passed messages through several layers, with a typical message packet read and written perhaps five times or more in its progress from source to destination.

In the new scheme, the message becomes a logical entity which has no existence as an individual object. This makes it possible for the bulk of the information in a message to be written once, when sent, and read once, when it is received at its final destination. Instead of individual messages, the program deals with packets containing many messages. The packet size is chosen to match the most efficient Myrinet transfer size, which is 1 KByte[5] in the current implementation.

Figure 4.2 shows the structure of a message packet. Each packet contains some header information, including a link field and the delivery time of the latest message in the packet, and a number of messages. Each message likewise contains a link field and delivery time. All messages in a packet will be delivered to the appropriate destination node by MPI, so sending that part of the address in the packet (let alone in each message) is redundant. The link fields might also seem redundant because they are filled in only at the destination, but including the empty fields eliminates the need to copy the messages from the packet to individual message buffers, and so improves overall performance.

[4]Excepting real-time I/O.
[5]See Table 1.1

Finally, the indexed addressing of messages to a (Cell, Compartment, Synapse) has been eliminated. Instead, the connection algorithm determines the address of the destination object on the receiving side and transmits it to the sender, which then uses it as the destination address field in messages. The receiver's message delivery code simply uses this address as a function pointer to call the destination object's message receiving function, thus replacing 12 bytes of indexes with a single 4-byte pointer, and reducing the delivery code to a single line. The cluster thus can be viewed as a very large segmented address space, where the node number is equivalent to the segment register and the 4 GByte physical address space of each CPU becomes the offset.

### 4.3.1 The Message-Passing Algorithm

The new `MessageBus` algorithm operates as follows.

At startup, the `MessageBus` for each node determines to which nodes it will be sending and prepares an empty outgoing packet for each node. It determines the permissible message delay for each sending and receiving node, prepares arrays in which the allowed and actual times will be stored, and creates ring buffers (`PendList` and `MsgList`) in which incoming

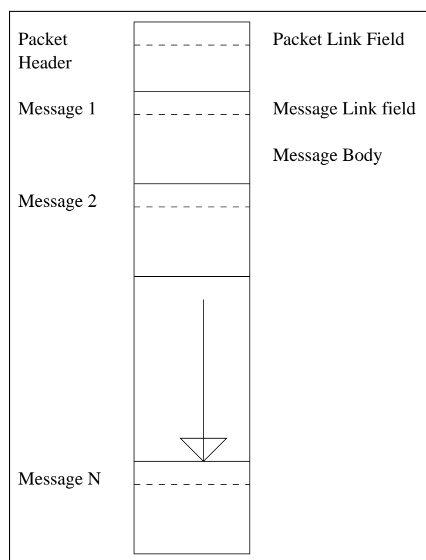


Figure 4.2: Structure of Message Packet.

packets and messages will be stored in linked lists until their delivery time. Figure 4.3 shows a diagram of the packet and message links.

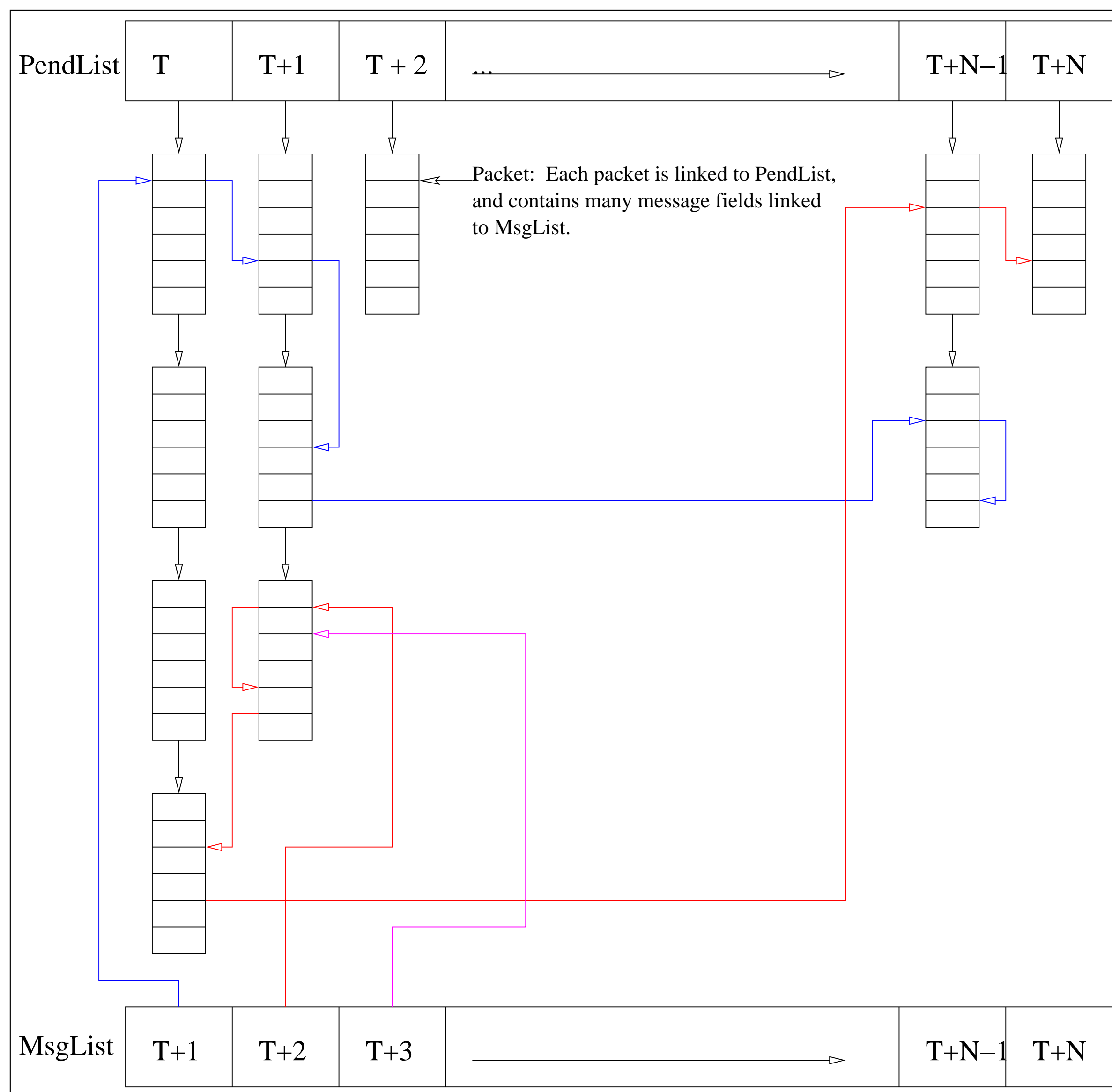


Figure 4.3: Schematic of MessageBus.

During the *DoCell* portion of each timestep, cells may fire. When one does, messages for each cell in the firing cell's *SendTo* list are placed in the destination nodes' outgoing packets, and the packets' *TimeSent* and *LastTime* fields are updated according to the synapse propagation time (which must be at least one timestep). When a packet is full it is sent, and

a new empty one is obtained from the packet pool. Meanwhile, the sent packet remains in an active state so that the non-blocking `MPI_Isend` function can be used, thus allowing overlap of communication and computation.

This process continues until all cells are processed, at which time the last packet has its `SYNC` flag set and is flushed. The `SYNC` flag informs the destination node that the sending node has completed the timestep. (If no messages are pending in the packet, it is sent empty, so that the destination node will still receive the `SYNC` flag.) The program then continues with the *DoReport* processing for the timestep while MPI/Myrinet is transferring the messages. In any significant model the computation time is several orders of magnitude larger than the packet transmission time, which allows most of the communication time to be effectively overlapped by computation.

The `MessageBus::ReceiveMsgs` function checks for incoming packets. When one is received, it checks the `SYNC` field, and updates the `NodeTime` entry for the sending node. It then places the packet in the slot of the `PendList` list that corresponds to the packet's `LastTime` field and walks through the messages in the packet, filling in the `Msg->link` field to add it to the linked list of messages in `MsgList` to be delivered at the `Msg->Time` timestep.

At each timestep, `DeliverMsgs` takes the messages in that list and delivers them to their destination compartments. The packet is meanwhile being held in the `PendList` (because messages are just fields in the packet). When the timestep reaches the current `PendList` entry, all messages in the packets in that entry have been processed (because the `PendList`is indexed by the `LastTime` field), and the empty packets can be returned to the packet pool.

### 4.3.2 Synchronization

Most of the computation time in the NCS program is used in computing the effects of synapse firings on the receiving compartments. The firing rates, however, are essentially random, being determined by the brain's reactions to stimulus. Therefore it can be expected that, regardless of how well the number of synapses is balanced between nodes, the actual amount of computation will vary both between nodes and over time.

As a consequence, one node, and probably not the same node at each timestep, will take the longest amount of time to finish its computations. If a simple end of timestep barrier is used for synchronization, then all the other nodes will be idle for some part of the timestep. Figure 4.4 shows an example of this idle time. Node 1 has (for the displayed timesteps) the heaviest load, and so displays little or no idle time (labeled `MessageBus::Sync` in the figure), while the others display more, with the amount varying between nodes and *between timesteps*.

This MessageBus implementation attempts to circumvent that situation. Recall from Section 1.4 that the electrochemical pulse from a firing cell propagates along its synapses at a relatively slow speed, so that the transmission time between the sending and receiving cells typically translates to several tens of simulation timesteps. Thus for each node there is an event horizon, which depends on the minimum synapse propagation time of the nodes with which it communicates. If this minimum time is $dt$, then nothing other nodes do at time $T$ can affect this node until time $T+dt$. Therefore, a barrier mechanism constructed to utilize this event horizon can allow some of the end-of-timestep idle time to be used. A node may simply continue to work until it reaches $T+dt$. Meanwhile, messages have continued to arrive from the other nodes, and unless the node is consistently under-loaded, these messages will contain `SYNC` flags indicating that their nodes have progressed to another timestep.

Synchronization now becomes a relatively simple matter. On initialization, a `NodeTime` array is allocated, with entries for each node from which the node receives messages. As `SYNC` packets are received, these times are updated. When the node reaches the end of each timestep, these NodeTimes fields are checked. If the other nodes are within the minimum time difference, then the node can proceed to the next timestep; if not, it must wait for more packets to be received and check again.

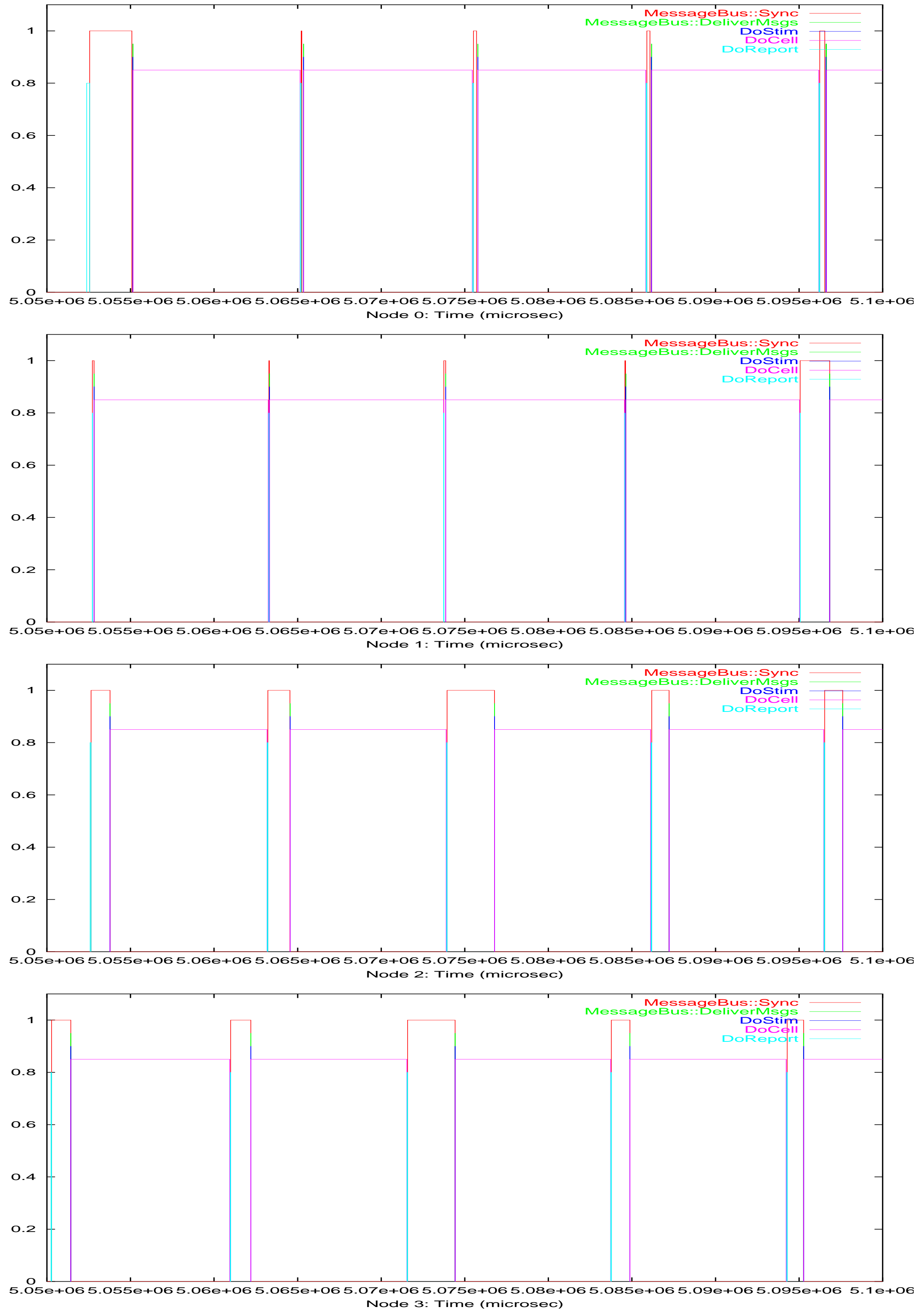


Figure 4.4: Idle Time Due to Load Imbalance.

# Chapter 5

# Results

In addition to the algorithmic improvements described in the preceeding chapters, many other optimizations have been made in the process of creating the NCS5 program. These are too numerous to describe individually. Indeed, most are obvious, and of no great theoretic interest. This chapter describes the cumulative effect of all optimizations, which fall into three categories:

- Sequential speedup, which increases the speed with which a particular compute node will process the part of the brain assigned to it.
- Parallel speedup. In the ideal case, a brain that takes time $T$ to run on a single processor would run in time $T/n$ when split among $n$ processors. Real programs virtually always achieve less than ideal speedup[1].
- Memory use, which determines the size of the largest model that will run in available memory.

## 5.1 Models Used In Testing

Several basic models were used in this performance testing. Table 5.1 summarizes their characteristics. While selected to test different aspects of NCS, they were not purpose-built for performance measurement but are instead modifications of models used for research in

[1] Neglecting such factors as cache effects.

our lab. It is therefore to be hoped that the measurements here will be a good approximation to real-world performance. These models are:

- 1Column - This model was used in the development of NCS3. It models a single column of three layers, each having two cell types, excitatory and inhibitory. Input is from an artificial pulse stimulus. It exhibits an unrealistically high cell firing rate.

- IVO - This is an adaption of the Intelligent Virtual Organism currently being developed. It was modified to contain a large number of similarly-sized small clusters, in order to allow distribution over the largest possible number of nodes. It was back-converted to NCS3 input format for comparison testing.

- BigIVO - This is identical to the IVO model, except that the cell counts have been increased by a factor of 5 and the number of synapses by a factor of 25 (since connection is $n^2$) in order to even out statistical fluctuations in load.

- AVI - Taken unchanged from ongoing research [1], it attempts to model parts of the audio and visual areas of the brain. With some 37.5 million synapses, it is too large to run on a single node.

| Model | Clusters | Cells | Synapses |
|---|---:|---:|---:|
| 1Column | 6 | 3,750 | 621,875 |
| IVO | 1,155 | 13,650 | 699,620 |
| BigIVO | 1,155 | 68,250 | 17,544,890 |
| AVI | 1,500 | 243,000 | 37,683,108 |

Table 5.1: Characteristics of Test Models

## 5.2 Sequential Performance Improvements

It is difficult if not impossible to define a simple performance metric for NCS. For a number of reasons, the time a particular NCS brain takes to process some input file is only a useful performance measure for that particular brain design and input.

One reason is that NCS defines many different components, which the user may include in fairly arbitrary proportions and connect in a large number of ways. Since the behavior of NCS is highly non-linear, these differences can result in large variations in processing time for models which might appear superficially similar.

More importantly, in a typical model the largest share of CPU time is devoted to synapse and spike processing. The spike rate depends on model parameters such as synapse conductance and connection patterns, *and also on the input being presented to the program*. As shown in Figure 5.3, the spike rate can thus vary considerably from timestep to timestep.

To further complicate matters, spike processing time is not necessarily even a linear function of the spike rate. There are two components to spike processing. First, some synapse receives each spike and processes it. There are many types of synapses, and this processing is different for each type. (Indeed, the differences are what defines the different synapse types.) All of them produce an identical result: the calculation of several factors that the owning compartment will apply to the synapse's post-synaptic conductance template.

The compartment then processes the templates of all incoming spikes. This processing is the same for all synapse types and continues over a number of timesteps defined by the length of the template. However, under some circumstances the compartment can optimize processing by combining all the spikes which it receives in a timestep, in which case the processing time no longer has a simple relationship to the spike rate.

On the basis of these issues, the approach taken here is to measure, on the same input, the performance of particular functional areas, or groups of operations with similar characteristics. Because the groups share common performance features, the effect of a change in the area on the whole program can be estimated. The area's speed change can be compared between program revisions: for example, if design A processes 1.0 million synapse firings per second, and design B processes 1.1 million, then design B has better performance.

The following are the functional areas measured:

- Overhead. This area encompasses all the functions that create the brain and its con-

nections and do other work associated with program initialization and termination. For most models it is dominated by the time needed to create the connections between cells. Processing time is largely independent of simulation length: simulating a few thousandths of a second or tens of seconds incurs the same overhead cost.

- Base cell and compartment. This is the time to process the simplest sort of cell. It also includes some overhead, such as stimulus input, that is not otherwise measured. Processing time is proportional to the number of cells in the brain.

- Channels. There are several types: the tables measure times for cells which have one channel for each type. Processing time is proportional to the total number of channels.

- Reports. Time is proportional to the number of items reported.

- Synapse and spike processing, as discussed above.

Table 5.2 shows the performance differences in these functional areas between NCS3 and NSC5,

Figure 5.1 shows the time usage of the components in a one simulated second run of the 1Column model described above. The cell firing rate[2] for this model is 282.4 per cell per second, well above the biologically-realistic range. Given the connectivity patterns specified in the model, this resulted in an average spiking rate of 161 million spikes per second.

Figure 5.2 shows the same information for a one simulated second run of the IVO model. The cell firing rate for this model is 64.4 per cell per second, much closer to the biologically-realistic range. Given the connectivity patterns specified in the model, this resulted in an average spiking rate of 45 million spikes per second.

The execution time of a model has a strong dependence on the spike rate, which will vary from timestep to timestep depending on the inputs presented to the program. The spik-

[2]Note the distinction between the firing rate and the spiking rate. Each firing cell sends spike messages to some number of other cells to which it is connected. This number varies from cell to cell, depending on the connection patterns specified in the input, and the particular random connections created in the connection phase.

| Item | NCS3 | NCS5 | Ratio |
|---|---|---|---|
| Overhead[a] | 1.897 | 294.167 | 155.1 |
| Base Cell/Cmp[b] | 0.020 | 3.035 | 153.6 |
| Channel[b] | 0.152 | 0.398 | 2.6 |
| Report[c] | 0.017 | 4.113 | 239.4 |
| Synapse, 0Hebb[b] | 0.031 | 0.383 | 12.5 |
| Synapse, +-Hebb[b] | 0.020 | 0.368 | 18.1 |

a) Seconds.
b) Millions of Objects Processed per Second
c) Millions of Values Reported per Second

Table 5.2: Preformance Ratios of Functional Areas.

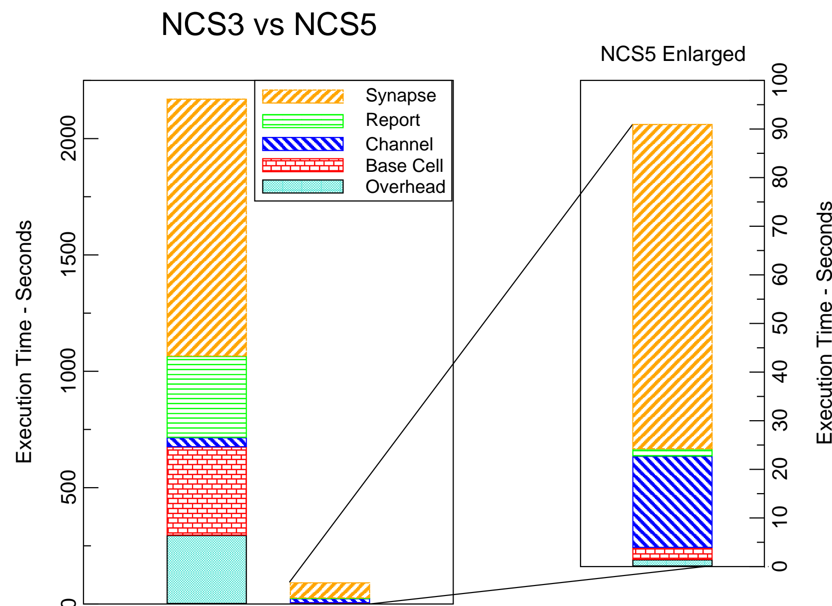


Figure 5.1: Share of CPU Time Used by Functional Areas, 1Column Model.

ing rate is a function of the cell firing rate and the connectivity; that is, each cell that fires produces a spike for each cell that it sends to. Recall from Section 1.4 that firing rates *in vivo* are observed to fall within a certain range, which a realistic model would expect to reproduce. Figure 5.3 shows execution time versus firing and spiking rates for a one such model.

While these rates depend in part on the input presented, they are also functions of model parameters, such as the synaptic conductance, which may be adjusted by the user. Figure

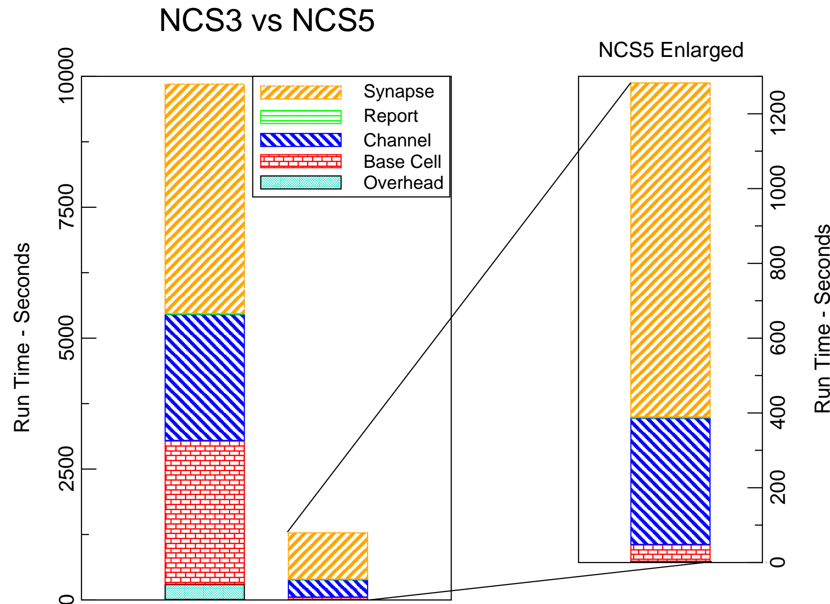


Figure 5.2: Share of CPU Time Used by Functional Areas, IVO Model.

5.4 shows how the execution time of the IVO model varies as the synaptic conductance is changed to produce cell firing rates across the biological range of 15-60 firings per cell per second. Note that the response to the changes is decidedly non-linear! Times are shown for both standard spike handling, and the optimized SAME_PSC version described in Section 10.

### 5.2.1 Parallel Performance

Although comparison of sequential performance is difficult, direct comparisons of parallel performance between NCS3 and NCS5 are, unfortunately, impossible. Due to a system crash and subsequent backup failure, all working parallel versions of NCS3 were lost shortly after the completion of [18], along with the data files used in its preparation. This section will attempt to make comparisons with some of the data reported there, but the reader should be aware that many factors that strongly affect performance, such as synapse firing and spiking rates, were not included in that report. Thus, for example, it reports in Figure 4.2 an execution time of some 13 hours for a 0.5 second simulation of 1.5 million synapses on 30 nodes, but

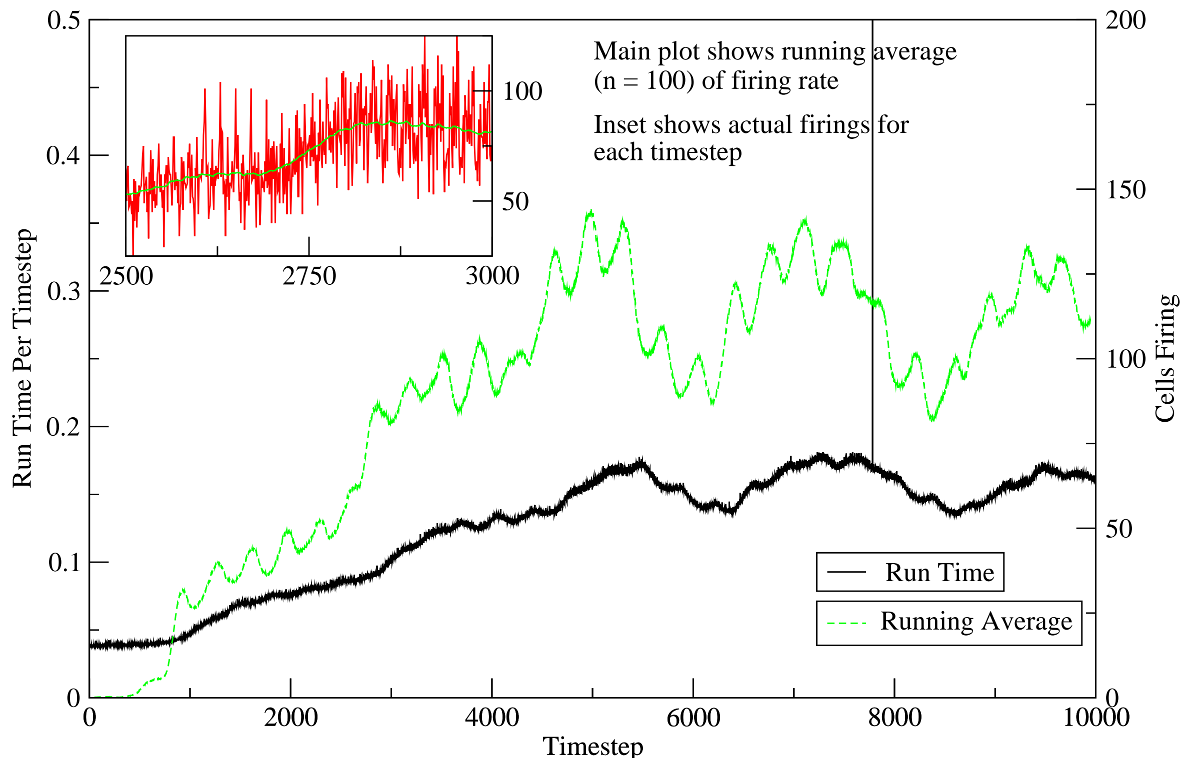


Figure 5.3: Processing Time and Firing Rate

there is no information as to what the spike rate was, and thus no way to make a meaningful comparison.

Note that all parallel test runs shown here were made with reporting turned off. During testing, several instances of the same model would be running simultaneously (on different nodes), leading to file name clashes.

Transmitting synapse firing information accounts for virtually all of the communication (other than the `SYNC` packets) between nodes. Forcing the cell firing rate to zero[3] should thus represent something of a base or ground state. Figure 5.5 shows run times for the IVO model with zero firing.

Figure 5.6 shows performance for the IVO model with a more realistic spike rate.

[3]In this case, by not supplying input to the brain

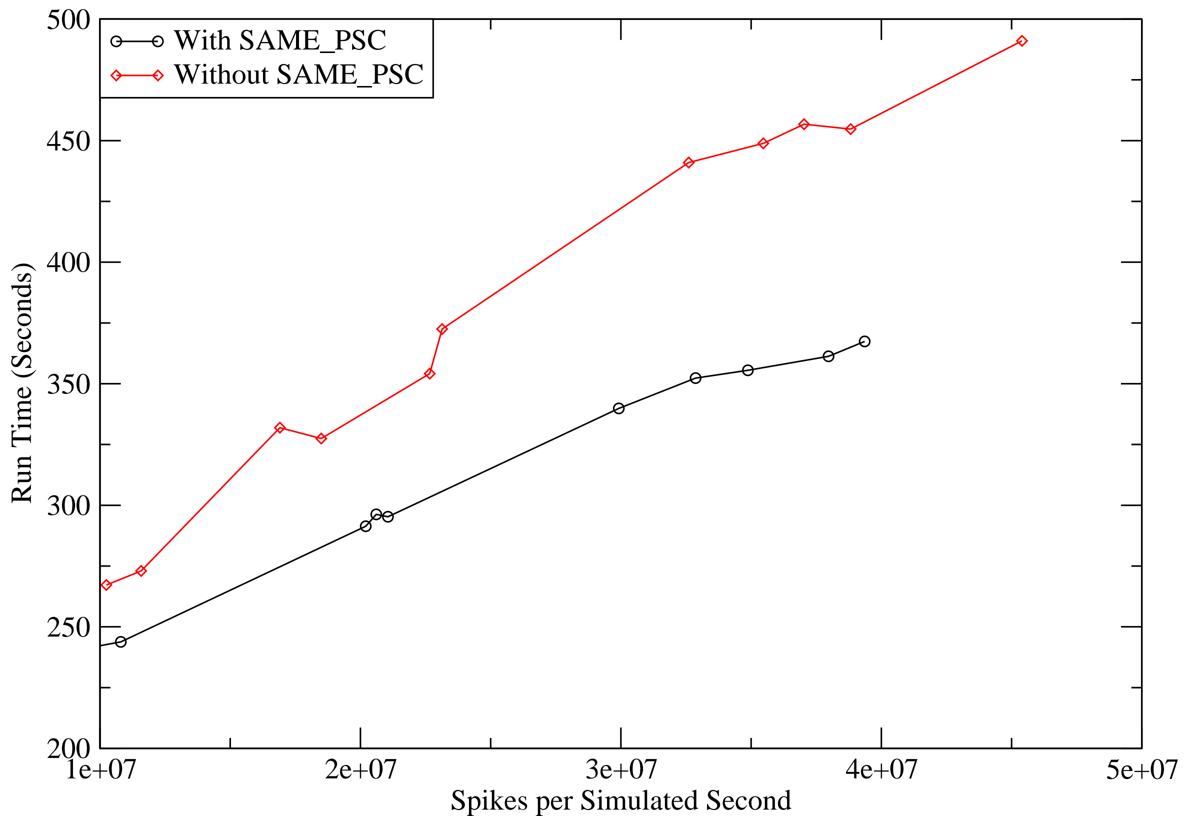


Figure 5.4: Variation of Execution Time with Spike Rate.

Recall from Section 1.5 that the Cortex cluster is composed of dual-processor motherboards. Figure 5.7 shows parallel performance for the same model when processors are allocated by twos - that is, the two CPUs on cluster node 0, then two on cluster node 1, *etc*. This gives somewhat poorer performance than when allocating one CPU on node 0, one on node 1, one on node 2, *etc*. This is counter-intuitive, since it might be expected that lower communication overhead between two CPUs on the same motherboard would result in a speedup, if anything[4]. Note also the change in slope when the processor count passes 40, and less-capable P3 processors begin to be used.

Figure 5.8 shows run times for the BIGIVO model. Notice that as each processor has more work than in the base IVO model, there is less statistical fluctuation in load, and hence somewhat better processor utilitization as the number used increases.

[4]We suspect this to be a side effect of scheduling with virtual processors enabled.

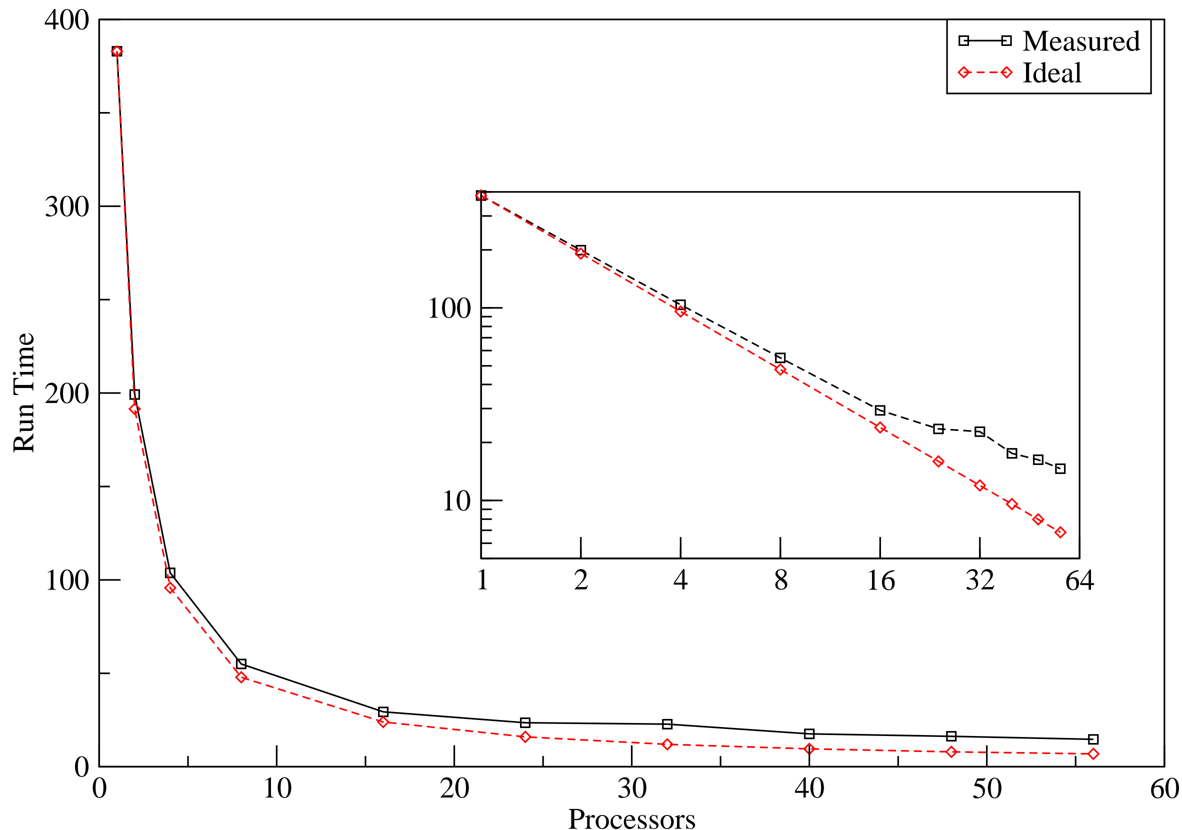


Figure 5.5: Parallel Run Times, IVO with No Firing.

And finally, Figure 5.9 shows run times for the AVI model. As each processor has yet more work than in the BigIVO model, processor utilitization improves to something very close to ideal speedup.

### 5.2.2 Virtual Processors

As noted in Section 1.5, each Pentium 4 processor in the cluster contains two virtual processors. Table 5.3 shows the result of test runs on the four virtual processors of one dual-CPU machine, *versus* on four distinct machines. For this example the run time using two real processors is less than when using two real and two virtual ones, so it appears that the virtual processors do not provide much, if any, increase in performance.

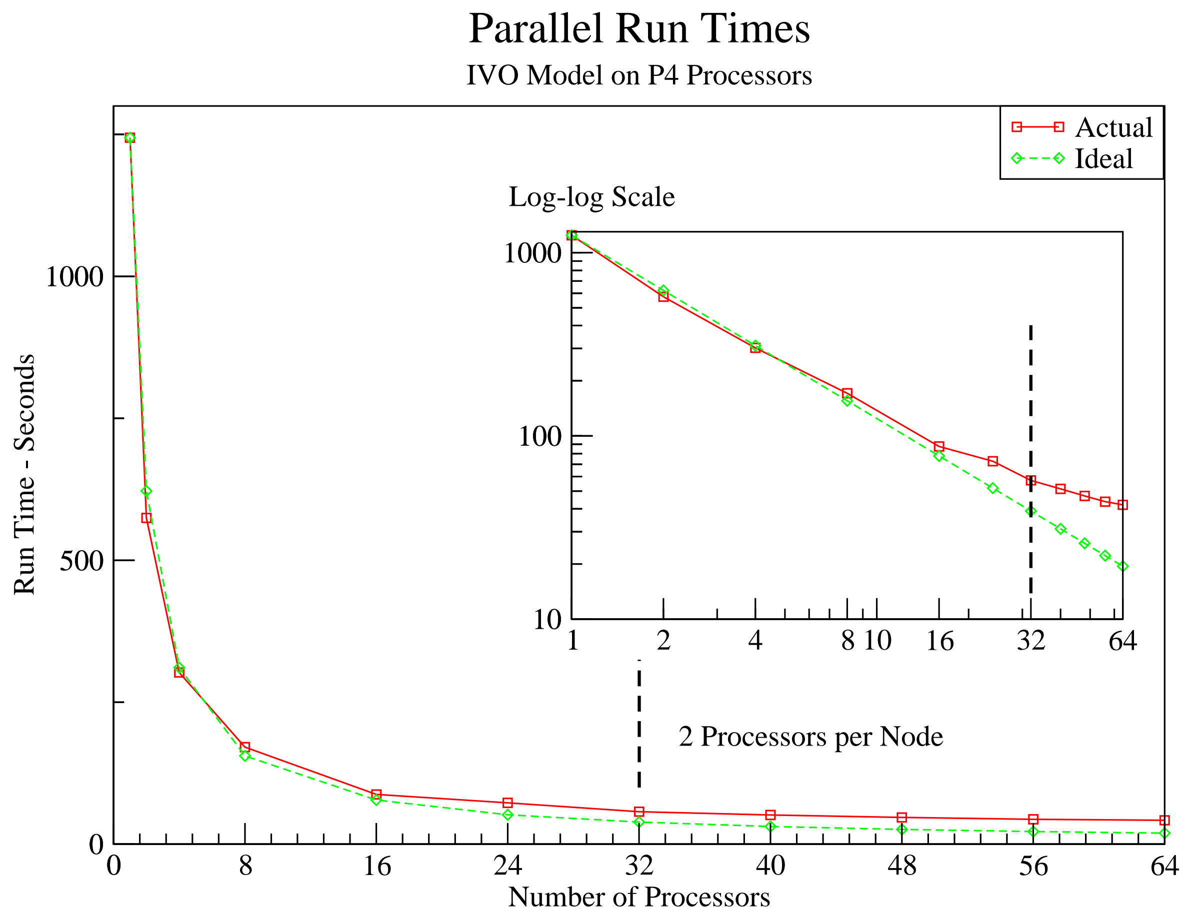


Figure 5.6: Parallel Run Times, IVO with Firing

| Condition | Run Time (sec) |
|---|---|
| 4 different nodes | 311.0 |
| 4 CPUs, 2 per node | 381.6 |
| 4 virtual CPUs on one node | 592.8 |
| 2 CPUs | 589.0 |

Table 5.3: Virtual and Dual CPU Performance

## 5.3 Memory Use

The original NCS3 program was limited in the size of the models it could handle. Although the precise limits have not been determined due to the excessive compute time required,

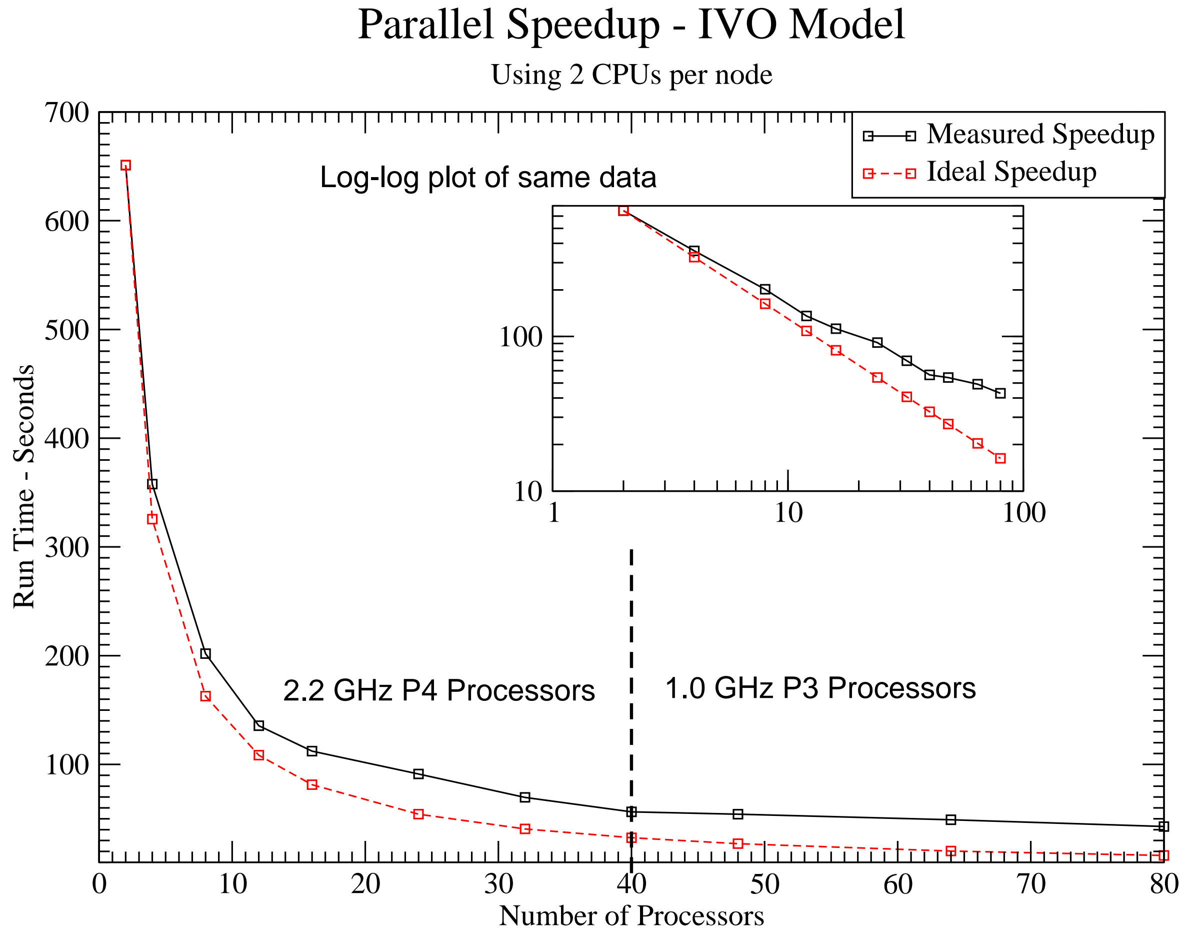


Figure 5.7: Parallel Speedup for IVO Model.

Wilson [18] cites as her largest case a brain with 1.5 million synapses distributed over 30 nodes (which took some 13 hours to process a 0.5 second simulation). The current code has demonstrated the ability to handle models of over 1.1 billion synapses.

There are three reasons for the difference in memory use:

- NCS3 created a global cell index with an entry for each cell and kept a full copy on each node. When a program was run on more than a few nodes, this index required more memory than did the actual brain. NCS5 creates a much smaller table, typically occupying only a few tens of megabytes.

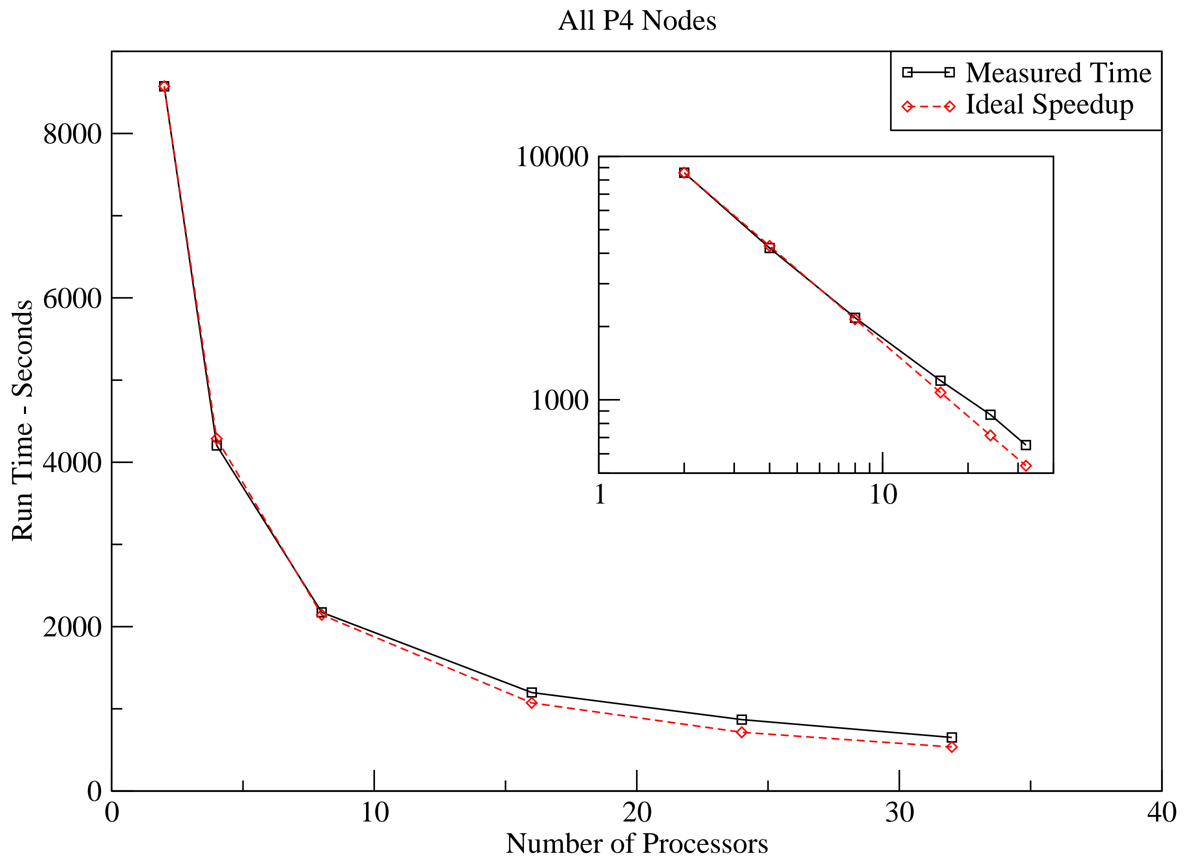


Figure 5.8: Run Times for BigIVO Model

- NCS3 pre-allocated many structures, such as message buffers, in amounts vastly larger than needed. For example, a message buffer was allocated for each synapse, even though any particular synapse might use its buffer only once in every several hundred timesteps.

- Finally, the NCS3 structures were usually much larger than actually needed.

Table 5.4 shows the amount of memory used by major brain components. For NCS3, memory requirements for a particular brain (exclusive of the global cell index) can be estimated from the number of cells, compartments, channels, and synapses. For NCS5, estimation is not so simple. Each synapse requires a `Synapse` and a `SendTo`, but because the other

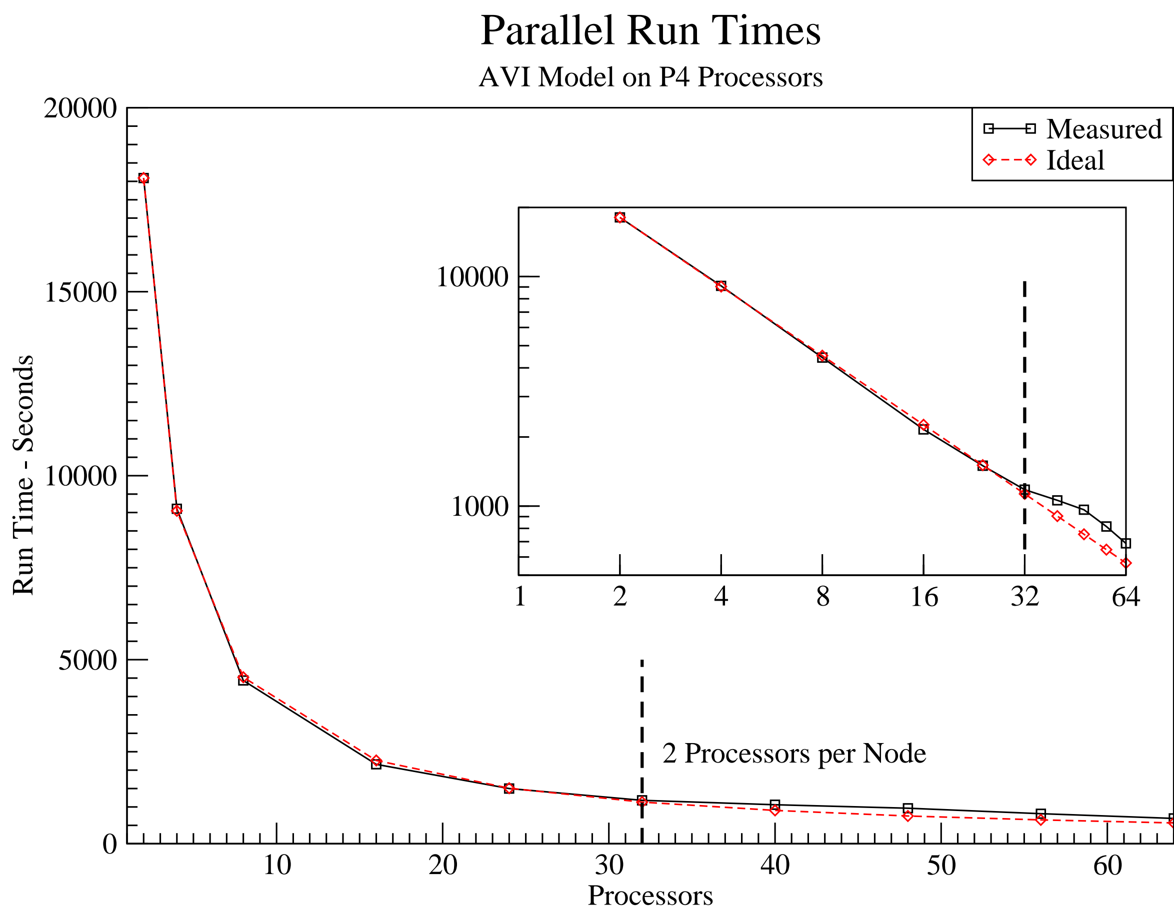


Figure 5.9: Run Times for AVI Model

synapse-related components are allocated as needed the memory actually used depends on the brain's spiking rate.

The total memory use of any particular model can easily be compared between versions. For example, the IVO model used for testing contains 13650 cells, 3 channels per cell, and 699,620 synapses[5]. NCS3 requires 800.117 MBytes of memory to run this model on a single node, while NCS5 requires only 99.559 MBytes. This difference is further exaggerated for larger models, which typically have a higher synapse/cell ratio.

Memory use distributes fairly well over nodes, as shown in Table 5.5. There is of

[5]Counted in NCS5. NCS3 creates a slightly different number of synapses, due to differences in the connection algorithms.

| Cell Components | NCS3 Bytes | NCS5 Bytes |
|---|---|---|
| Cell | 112 | 16 |
| Compartment | 548 | 284 |
| Channel | 392 | 252 |
| Synapse | 108 | 44 |
| CompartmentDelay | 12 | - |
| CellCompConnect | 20 | - |
| SynapsePSGOutgoing | 12 | - |
| SynapseDelay | 12 | - |
| Sendto | - | 12 |
| ActiveSynPtr | - | 20 |
| Message | 60 | 20 |

Table 5.4: Memory Used by Components - NCS3 *vs* NCS5

course some overhead for the information the program uses to coordinate actions among many nodes, but at 64 nodes, this overhead is only about 58 Mbytes per node, in a brain that occupies some 5 GBytes.

| Nodes | Brain | Dynamic | Total |
|---|---|---|---|
| 2 | 3497.2 | 1577.4 | 5074.6 |
| 4 | 3556.5 | 1578.9 | 5135.4 |
| 8 | 3674.2 | 1580.5 | 5254.7 |
| 16 | 3909.4 | 1584.6 | 5494.0 |
| 32 | 4377.7 | 1599.9 | 5977.6 |
| 64 | 5324.3 | 1643.9 | 6968.2 |

Table 5.5: Memory Use by AVI Model (MBytes).

The memory use shown has two components. Brain memory is that used to create all the brain objects and supporting structures (the overhead), and is the amount of memory used at the start of thinking. Dynamic memory is that allocated during the thinking process. It holds message packets, active synapse structures, and the like. The amount of dynamic memory required is not entirely predictable, as it is dependent on variables such as the spike rate, but for any particular input it will stabilize when the brain has become fully active.

# Chapter 6

# Conclusions and Future Work

The work covered by this thesis has:

- Significantly improved the performance of the NCS program.
- Demonstrated the capability of simulating realistic brains containing upwards of $10^6$ cells and $10^9$ synapses.
- Demonstrated that model size and performance scale linearly over the number of processors available for testing.
- Created a number of useful profiling and performance measurement tools.

## 6.1 Performance - Time

Two separate areas of performance, sequential and parallel, have been addressed in this work.

The issue of sequential performance is complicated by the strong dependence of performance on spike rates, which in turn is dependent on both the design of a particular model and the input which is presented to it. There is a base performance, the time needed to process the cells and their components in the absence of any spikes, and the spike processing performance, which is complicated by factors such as synapse learning. Adding to the complication are ongoing changes to input formats and the detection and repair of a number of bugs that substantially affect results. In consequence, any performance comparison is both

difficult and somewhat arbitrary. Even when the same model is created in the different formats required by the old and new code, running that model on identical inputs will produce different outputs and spike rates.

Various brain components can be tested and compared, and all of these show some degree of speedup, ranging from a little over twice for channels, to two orders of magnitude for the compartments and reports. The total effect these improvements will have on a particular brain will depend on the proportions in which the corresponding components are used.

Parallel performance comparisons are more problematic yet, because no working versions of the parallel NCS3 code exist to compare against. The results reported in [18] are the only available standard, but the information needed to make an accurate comparison is not reported.

However, it is known (Section 4.1.4 of [18]) that a model containing 1.5 million synapses, run for a simulated time of 0.5 second, took approximately 50,000 seconds execution time on the original 60 P3-processor Cortex cluster. For comparison, the BigIVO model tested here, with some 17 million synapses, requires about 870 seconds for one simulated second on 24 P4 processors, while the AVI model with 37.6 million synapses requires 1490 seconds for 0.5 simulated second on the same 24 processors. In the absence of a better standard of comparison, this would suggest that the overall performance improvement from this work is between two and three orders of magnitude.

## 6.2 Performance - Memory

In addition to processing speed, optimization of memory use is critical to the lab's planned research. Our cluster provides a mere 256 GBytes of RAM, but some of the models our team would like to run require billions of synapses, each one requiring some memory to hold its individual attributes and state. Here the comparison between versions is on somewhat firmer ground. Although the 1.5 million synapse model is the largest reported for NCS3, NCS5 has demonstrated the ability to run models containing at least 1.1 billion synapses, an

improvement of nearly three orders of magnitude[1].

## 6.3 Analysis

It is clear that the work described here has produced significant improvements in both speed and size.

In computer science, it has lately become something of a truism that code optimization (or at least optimization done by programmers rather than compilers) is wasted effort: if more performance is needed, one simply buys faster processors. This thesis demonstrates its falsity, at least for computationally significant applications.

Consider that during the course of this work, the size of the Cortex cluster was doubled, with the new processors being about 3 times faster than the original. This was done at a cost of some $200,000 and resulted in about a fivefold increase in performance and a doubling of maximum model size. Over the same period, the programmer effort invested in optimization would have cost perhaps half that amount at market rates, and resulted in at the most conservative estimate a tenfold increase in speed, and a thousand-fold increase in maximum model size.

Much of the difference in performance, especially in memory use, between the two versions may be traced to a too-strict adherence to an object-oriented design philosophy in the the original code. When a program is seen as a collection of abstract objects, divorced from their actual machine representation, it becomes difficult to keep in mind that those objects and their operations will always incur some cost in machine cycles and memory, and *that cost does not necessarily bear any relation to the representation of the object in the source code.*

It is also interesting to compare the source code sizes of the two versions. Despite the addition of a number of new features to the program, the number of source code lines, as shown in Table 6.1, has shrunk by nearly 45%.

[1] Since these models were tested, memory use has been further improved.

| Item | NCS3 | NCS5 | Ratio |
|---|---|---|---|
| Total Lines | 26720 | 16912 | 0.63 |
| Code Lines | 19801 | 11114 | 0.56 |
| Blank Lines | 4880 | 3172 | 0.65 |
| Comments | 2039 | 2626 | 1.28 |

Table 6.1: Code Size Comparison

## 6.4 Future Work

Despite the significant performance increases in the course of this work, there are still areas - not limited to performance - where further improvements might be made. Some of these are:

- `Channel` and `Synapse` objects. Each of these have many variants, differentiated by internal logic within the single `Channel` or `Synapse` class. The objects thus contain internal variables that are used in some variants, but not others. Implementing the variants as sub-classes which inherit from a base `Channel` or `Synapse` class will thus save memory, and might improve performance.

- Synapse Model. As is evident from Figures 5.1 and 5.2, spike processing now uses by far the greatest share of processing time, much of which is incurred by compartments iterating through long lists of PSC templates. A model that uses a computation rather than the template list processing might well be faster and use less memory.

- Channel Model. Channel processing consumes the second largest share of processing time. Unfortunately there seems little scope for improvement within the current channel model, as channels are little more than a simple calculation of a few exponential and power equations. Lookup tables might be a faster alternative.

- Threaded Message Receive. Currently incoming messages are processed only at certain points in any timestep. Efficiency might be improved if message reception was

threaded out, so that each message packet would move from the MPI subsystem to program space as soon as it arrives.

- Distribution. It should be possible to distribute cells to the various nodes in such a way as to minimize inter-node message traffic and its associated overhead. Likewise, if clusters can be assigned to nodes in such a way that a degree of pipelining is possible, then the MessageBus will allow more overlap of computation, and throughput will be improved.
- Load Balancing: At present, when some area of the code has been improved, it is necessary to measure the new performance weights manually and apply them to the code. It should be possible to automate this process.
- Consistent random number generation: For improved biological realism, many cell and synapse parameters can be specified with a random variation. Parallel random number generation currently is not consistent, so that the output of the same model will show some random variation when run on different numbers of nodes.

## 6.5 Finally...

As noted in the first chapter, the human brain contains some $10^{11}$ cells, with an estimated $10^{14}$ synapses. Currently we can simulate approximately $10^{6}$ cells and $10^{9}$ synapses, at a rate of perhaps $10^{4}$ seconds of computation to each second of real time. Thus our program and cluster, for all that it is close to state-of-the-art computing technology, is capable of simulating only about one billionth of the activity of a human brain.

# Bibliography

[1] J.L. Blake and P.H. Goodman. Speech perception simulated in a biologically-realistic model of auditory neocortex. *Journal of Investigative Medicine*, 2004.

[2] B.W. Connors, M.J. Gutnick, and D.A. Prince. Electrophysiological properties of neocortical neurons in vitro. *J. Neurophysiol.*, 48(6):1302–1320, 1982.

[3] Intel Corporation. Using the rdtsc instruction for performance monitoring. `http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM`, 1997.

[4] Ed. Dale Purves. *Neuroscience*. Sinauer Associates, Sunderland, Massechusetts, 1997.

[5] Free Software Foundation. Bison user manual. http://www.gnu.org///bison.html (11/29/03).

[6] Free Software Foundation. Flex user manual. http://www.gnu.org///flex.html (11/29/03).

[7] Free Software Foundation. gdb user manual. http://www.gnu.org///debug/gdb.html (11/29/03).

[8] M.M. Kellog, H.R. Wills, and P.H. Goodman. A biologically realistic computer model of neocortical associative learning for the study of aging and dementia. *J. Investig. Med.*, **47**(2), February 1999.

[9] J. C. Macera, P. H. Goodman, F. C. Harris, Jr., R. Drewes, and J. Maciokas. Remote-neocortex control of robotic search and threat identification. *Robotics and Autonomous Systems Journal*, November 2003. Accepted November 2003.

[10] Juan Carlos Macera. Design and implementation of a hierarchical robotic system: A platform for artificial intelligence investigation. Master's thesis, University of Nevada, Reno, December 2003.

[11] James B. Maciokas. *Towards an Understanding of the Synergistic Properties of Cortical Processing: A Neuronal Computational Modeling Approach*. PhD thesis, University of Nevada, Reno, August 2003.

[12] Myricom Inc. Creators of Myrinet. `http://www.myrinet.com`, June 2002. 325 N. Santa Anita Ave. Arcadia, CA 91006.

[13] University of Oregon. Tau portable profiling. http://www..uoregon.edu//paracomp/tau/tautools (12/10/03).

[14] Goodman P.H., E.C. Wilson, J.B. Maciokas, F. C. Harris, Jr., S.J. Louis, A. Gupta, and H.J. Markram. Large-scale parallel simulation of physiologically realistic multicolumn sensory cortex. Tech Report 01-01, http://brain.unr.edu/publications/goodmanNIPS01final.pdf.

[15] Luis De Rose, Ying Zhang, and Daniel A. Reed. Svpablo: A multi-language preformance analysis system. `http://citeseer.nj.nec.com/derose98svpablo.html (12/10/03)`, 1997.

[16] K.K. Waikul, L. Jiang, F. C. Harris, Jr., and P.H. Goodman. Implementation of a web portal for a neocortical simulator. *CATA 2002 Proceedings*, 2002.

[17] H.R. Wills, M.M. Kellogg, and P.H. Goodman. Cumulative synaptic loss in aging and alzheimer's dementia: A biologically realistic computer model. *J. Investig. Med.*, **47**(2), February 1999.

[18] E. Courtenay Wilson. Parallel implementation of a large scale biologically realistic neocortical neural network simulator. Master's thesis, University of Nevada, Reno, August 2001.

[19] E. Courtenay Wilson, Phillip H. Goodman, and Jr. Frederick C. Harris. Implementation of a biologically realistic parallel neocortical-neural network simulator. Proc. of the $10^{th}$ SIAM Conf. on Parallel Process. for Sci. Comput., March 2001.

[20] E. Courtenay Wilson, Frederick C. Harris, Jr., and Phillip H. Goodman. A large-scale biologically realistic cortical simulator. Proc. of SC 2001, November 2001.