CMPE-655 Spring 2017 Assignment 1: Compartmental Hodgkin-Huxley Neuron Model, Parallel Implementation with MPI

Rochester Institute of Technology, Department of Computer Engineering
Instructor: Dr. Shaaban (meseec@rit.edu)
TAs: Kartik Patel (kpp1990@rit.edu) and
Karishma Reddy (kgr2269@rit.edu)
Due: Monday, March 27 , 2017, 23:59:00, Project designed by: Dmitri Yudanov

Contents

Introduction

The Message Passing Interface (MPI) is commonly used to define and implement parallel computations on message-passing computer architectures. The MPI definition is extremely large, and can be intimidating when first encountered, but once the initial learning curve is overcome, MPI can be a very useful tool for implementing parallel programs. MPI, like any good API, allows a developer or researcher to write high-level code without needing to worry about implementation details.

With this power, however, comes the responsibility of writing proper parallel code. MPI may fail to improve a problem's performance, not because of lack of parallelism in the problem, but due to programmer error.

Objectives

For this assignment, you will be exploring the effects of model parameters on a simple compartmental Hodgkin-Huxley neuron model. You will be given a sequential implementation of the neuron model, and it will be your job to parallelize the given code using MPI. You will then use your parallel implementation to learn more about the model itself, to learn more about the effect certain parameters have on execution time, and to learn more about parallel programming in general.

Specifically, you will be recording the effect of dendrite length (*i.e.*, the number of *compartments* in a *dendrite*) and the number of dendrites on the execution time and integration response of a neuron's *soma*. You will also be exploring the effect of parallel implementation and load balancing on computation time and parallel vs. sequential speedup.

Course Website

http://mps.ce.rit.edu

This document, other files, and information necessary for the completion of this assignment can be found at the above website. This website is where you will find the sequential implementation of the project and it is where you will find information about:

- Using MPI (programming, compiling, and running MPI programs)
- Connecting to and using the computer engineering department's cluster
- Project submission process

Required Measurements

To complete this assignment, you will need to record data for the following tables, and answer the questions associated with those measurements in paragraph form in the Results/Data Analysis section of the report. Bulleted lists of answers will result in point deductions.

Care should be taken to ensure that your program will not run on an oversubscribed node where the number of worker threads exceed the number of available real cores; running all 13 threads on the same node does not give valid results. **This can be easily accomplished by using the SLURM scheduler properly.**

Question set 1:

Number of Slaves	Dendrites	Compartments	Exec Time (sec)	Speedup
0, sequential	15	10		1
5 (mpirun –np 6)	15	10		
12 (mpirun –np 13)	15	10		
0, sequential	15	100		1
5 (mpirun –np 6)	15	100		
12 (mpirun –np 13)	15	100		
0, sequential	15	1000		1
5 (mpirun –np 6)	15	1000		
12 (mpirun –np 13)	15	1000		

Table 1: Effect of Dendrite Length on Computational Load

- What effects do you observe from the parallel implementation? Explain why these effects are present.
- Why is speedup not the same as the number of parallel processes?
- What can you tell about the spiking patterns of the *soma* graphs relative to the dendrite length?

Question set 2:

Number of Slaves	Dendrites	Compartments	Exec Time (sec)	Speedup
0, sequential	15	10		1
5 (mpirun –np 6)	15	10		
12 (mpirun –np 13)	15	10		
0, sequential	150	10		1
5 (mpirun –np 6)	150	10		
12 (mpirun –np 13)	150	10		
0, sequential	1500	10		1
5 (mpirun –np 6)	1500	10		
12 (mpirun –np 13)	1500	10		

Table 2: Effect of Number of Dendrites on Computational Load

- What effects do you observe from your parallel implementation? Explain why these effects are present.
- What similarities/differences do you see when comparing to the data from Table 1? Explain.
- What can you tell about the spiking patterns of the *soma* relative to the number of dendrites? Compare to the data obtained from the experiments used to generate Table 1.

Question set 3:

#	Number of Slaves	Dendrites	Compartments	Exec Time (sec)
1	10 (mpirun –np 11)	30	100	
2	10 (mpirun –np 11)	33	100	
3	10 (mpirun –np 11)	36	100	

Table 3: Test for Load Imbalance

- Compare and comment on the execution time of experiment #1 and #3 relative to #2 in Table 3. What does this tell you about how effective your program is at balancing computational load?

Question set 4:

#	Number of Slaves	Dendrites	Compartments	Exec Time (sec)
1	5 (mpirun –np 6)	5	1000	
2	5 (mpirun –np 6)	6	1000	
3	5 (mpirun –np 6)	7	1000	
4	5 (mpirun –np 6)	1199	10	
5	5 (mpirun –np 6)	1200	10	
6	5 (mpirun –np 6)	1201	10	

Table 4: Effect of Load Imbalance

- Compare and contrast the effect of load imbalance in experiments #1-2 and experiments #3-4 in Table 4. You may find it useful to compare in terms of % of execution time. How would you improve the worst case among these results?

Neuron Model Background Information

Further information is available in the included Appendices. Provided here is a basic explanation of the neuron model used in this assignment.

The Hodgkin-Huxley model of a neuron is a Nobel Prize winning model that gives an explanation of basic neuronal operations. From an engineering perspective a neuron is a biological electrically conductive and excitable device that performs integration of incoming electrical signals and outputs a response signal that can be characterized based on frequency, amplitude, waveform shape and other metrics. In this project, we explore some basic principles of neuronal signal integration and we simulated a simple model of a neuron.

Figure 1a displays a typical morphology of a neuron. Multiple wires called *dendrites* provide a path for incoming signals. Dendrites are connected to the cell body, called *soma*.

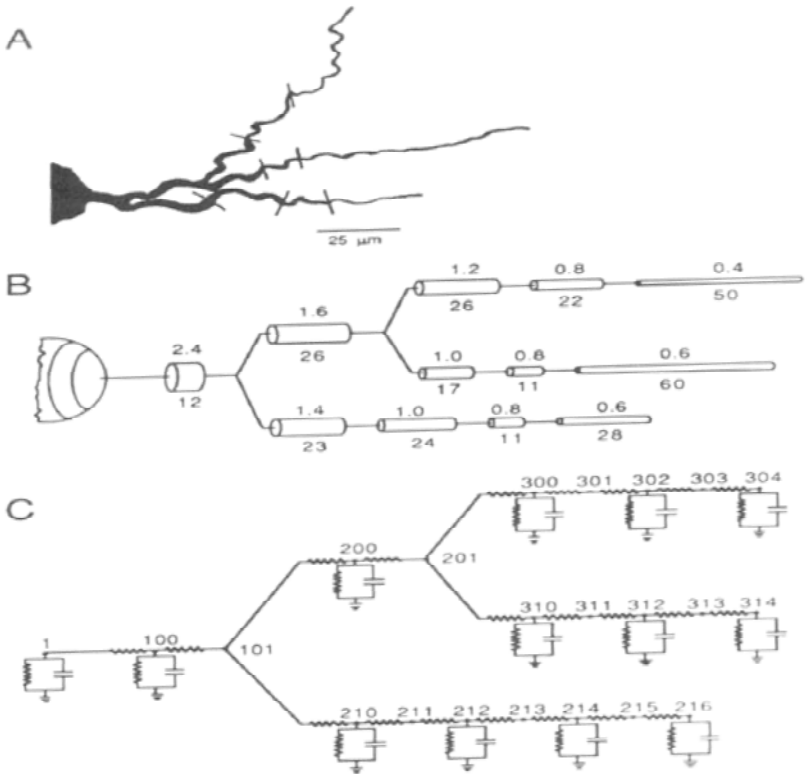

Figure 1: Compartmental model of a neuron (taken from "Methods in Neuronal Modeling. From Synapses to Networks," Christof Koch and Idan Segev 1989)

In the neuronal world, cytoplasmic (internal to a cell) and extracellular parts are separated by a membrane. Located on this membrane "skin" are various sophisticated devices such as ion channels and ion pumps. These devices maintain the cytoplasmic side at a lower electrical potential relative to the extracellular side. Thus, inward current is easily generated in the places where the membrane has openings. These openings usually occur as a response to an increase in the concentration of neurotransmitters, which are released by other neurons that "touch" the dendrites of a neuron in places called *synapses*. In this project, we do not model synapses. Instead, we directly inject randomly varying current at the tips of the dendrites.

As a result of either synaptic or artificially injected current, the transverse electrical current flows through the dendrites from higher to lower electrical potential towards the soma. Some current is lost due to the resistive nature of the dendrites and some leaks back to the extracellular side of the neuron via the neuron's membrane. At the same time, however, transverse current that flows inside the dendrites may also get amplified by active devices (*e.g.*, ion channels). We do not model ion channels in the dendrites in this project, but we do model them in the soma as a conductance.

If the electrical potential of a soma reaches a certain threshold value, a rapid current influx is generated due to ion channels that are designed by nature to open a "hole" (*i.e.*, increase conductance) in the neuron membrane in response to the potential. As a result, a positive (references to the extracellular

side) potential spike called the *action potential* is produced. This spike gets picked up by *axon* (a long extension of the soma) and is delivered to other neurons, but, again, we do not model an axon in this project.

Each dendrite can be modeled with multiple *compartments* that represent segments of a cell with constant transverse (*i.e.*, series) resistance and an associated lateral (shunt) membrane capacitance (see Figure 1B/C). Thus, a dendrite is nothing but a non-inductive leaky transmission line (for further information, see Appendix C). Multiple lines can be assembled into a binary tree (Figure 1C). In our model, a simpler topology is used: all dendrites are connected directly to the soma without any connection to each other, plus, each dendrite in the model has the same number of compartments, and each compartment has the same parameters. However, we do randomly vary the current injected at the tip of each dendrite.

If we increase the number of compartments in a dendrite, we increase its total series resistance, decrease the membrane (shunt) resistance, and increase the dendritic membrane (shunt) capacitance. Thus, a larger portion of current that we send from the tip of the dendrite to the soma is lost, and we also increase computational load for the dendrite, because we have more compartments to compute per dendrite. To compensate for the loss of current, neurons have multiple dendrites that have synapses with incoming neurons. Again, though, the more dendrites, the more computational load. Thus, if we wish to execute this system using MPI, we must consider the effect both of these parameters have on computation load.

Code for Sequential Model

The provided code implements a model based on a combination of the concepts described in Appendix C, and the Hodgkin-Huxley neuron model from ModelDB developed by Dr. Stewart and Dr. Bair of Oxford (see http://senselab.med.yale.edu/ModelDB/ShowModel.asp?model=117361 and Appendix D for more information). The integration method used is a Runge Kutta 4^{th} order stepper (RK4). To understand the functionality of the code, it should be sufficient to read the comments in the provided header and implementation files.

There is no requirement that you understand how the neuron model works, however, if you want to understand the concepts behind the model, please take a look at Appendices A-D, with an emphasis on Appendix A and C.

To find out where and how to exploit parallel computation, it is sufficient to concentrate your attention on the “Main computation” code in “seq_hh.c” and the flowchart in Figure 2. The implementation file “lib_hh.c” and its associated header file provide the functionality of the model.

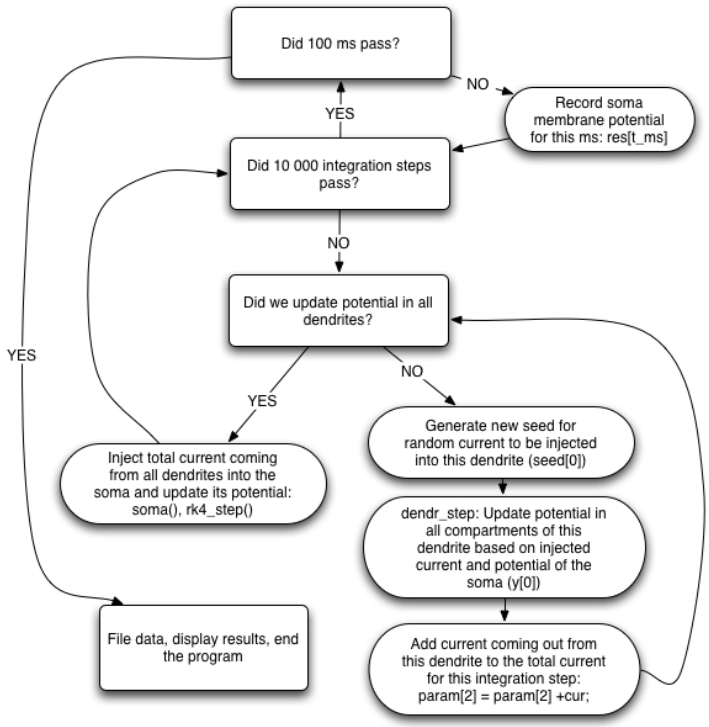

Figure 2: Flowchart for Main Computation Loop

Project Requirements

In addition to all requirements on the course website (http://mps.ce.rit.edu/meta/submissions.php), the project is also subject to these rules:

- Your report must answer all questions posed in the above "question sets".
- For results to be submitted, no parameters in the model may be altered, except the ones specified (*i.e.*, the number of dendrites and compartments). You are, of course, free to explore the effect of other parameters on your own (*e.g.*, injected current, resistances, capacitances, the integration time step, etc.).
- You must compile your results into a well written report that includes an abstract, detailed design methodology, results and analysis, and a conclusion. Answers to the questions should appear with the figures that are relevant in the results and analysis section. Adhering to standard CE report expectations is a good idea.
- All time recordings should be taken in the same way as they are in the provided sequential code.
- The number of dendrites that your parallel program has to accept must be at least equal to the number of slave processes. If the number of dendrites is equal to the number of slaves, then the master will not operate on the dendrites. In all other cases, the dendrites will be processed by both the master and the slaves.
- Your program must accept an arbitrary number of compartments and dendrites (subject to the previous rule).

- Your program must use a master/slave paradigm, where the *soma* is the master.
- The mapping of dendrites to parallel processes must be done in a way that provides the best load balance for any combination of dendrites/compartments.
- The filename of your parallel implementation must be "mpi_hh.c".
- Your program may only use basic sends and receives for communication (*i.e.*, you may not use MPI_Bcast() or MPI_Reduce()). This is to ensure that you understand the different communication types and the parameters needed by the MPI_Send() and MPI_Recv() functions.
- The user interface and I/O must be the same as in the sequential program. This means that there should be no difference for the user between running the provided sequential code and your program, with the obvious exception that your program will use `mpirun'.
- Your program must work on cluster.ce.rit.edu, regardless of where you develop it. For all measurements, you must use the cluster.
- Coding style and comments must be at the same level as in the provided code, including proper indentation.
- Make sure to read the README files and mps.ce.rit.edu website.
- Don't run simulations on the head node, and don't leave zombie processes running. Recalcitrant offenders will be penalized. If you use SLURM correctly, everything will work fine. See project/README and tips below for details and cleanup tips.

Grading

- 50% for correct program execution
- 15% for implementation and coding style
    - Implementation – Does the code look like it will run correctly?
    - Coding Style – Readability and commenting
- 35% for the report, including program performance

- Test your code before submitting. If your submission does not compile, no credit will be given for program execution. Corrected submissions will be accepted and graded according to the standard late policy.

Tips

- Start early! Some of the required measurements take well over an hour to run on an unloaded cluster. Even if no one makes a single mistake, even if no one consumes too many resources with a buggy program, even if everything works perfectly, the performance of the cluster will drop significantly if too many people are using it. The penalty for late submissions is severe, so get working as soon as possible (see http://mps.ce.rit.edu/meta/latepolicy.php).
- Start early (again)! Some tasks require 13 processes to run. Our cluster has 144 cores, so only 11 students can run 13 processes each at a single time. If, while in the middle of running a task, a

node becomes overworked, you run the risk of your results being meaningless (and you may be penalized for not running it right, especially if you don't show that there was a problem).

- Start early (one more time)! If I had a golden dollar for every person that will ignore this advice, I would have a very shiny change dish. If *you* start early, you're likely to not need to worry about cluster load, because no one else heeded the advice to start early.
- Develop at home. OpenMPI is a great implementation of MPI. If you're using Mac OS X, it should already be installed if you've installed the developer's package. If you're using Ubuntu, there is an OpenMPI package available from the Ubuntu package repositories; see the course website for additional information regarding Windows development. Developing at home has many benefits:
    - If you install an OpenMPI distribution, it will come with fantastic documentation of all the MPI functions (*i.e.*, it comes with man pages for all functions). This is extremely useful since documentation on the web can be very poor, incorrect, or just hard to find.
    - If you mess something up, you will not bring down the cluster. The importance of this cannot be overstated. Developing on the cluster brings the requirement that you stay on top of any zombie processes you may create. These processes will hog resources until eventually no one can use the node on which those processes exist.
    - You can use whatever text editor/development environment you like.
    - Of course, if you start very early, you would usually pretty much have the cluster to yourself, and all the above would be pretty much redundant. See a pattern here?
- **You are prohibited from running any jobs on the head node. Compilation and development are the only things that you are allowed to do on the head nodes.**
- As you develop your code, you should be using the resources that that are available on the cluster, such as the debug partition. It is prohibited from using *mpirun* on the head nodes, regardless of the duration of the job. **You must submit all of your work using to SLURM using the sbatch command, regardless of the number of processes that are required.**
- Use the scancel –u *username* or orte-clean command if you get into trouble. This will kill all of your running processes, on all nodes of the cluster. It ensures that you're not leaving anything behind taking up resources that other people will need. See (http://mps.ce.rit.edu/guide/tutorial.php#grim) for a fuller explanation. **If you feel that something is terribly wrong, contact your TAs as soon as possible.**

Credit

This project, the neuron model explanation, and the project's original source code are all the work of Dmitri Yudanov, a former RIT student.