CMPE-655 Multiple Processor Systems

Assignment One

Compartmental Hodgkin-Huxley Neuron Model, Parallel Implementation with MPI

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students. Other than code provided by the instructor for this exercise, all code was developed by me.

Cole Johnson, Mark Danza, Kevin Clampitt
March 20, 2025

Lab Section: L1
Instructor: Dr. Shaaban
TA: Chris Lenhard
Jake Pregitzer

Lecture Section: 01
Lecture Instructor: Dr. Shaaban

Abstract

The purpose of this exercise was to emulate the Hodgkin-Huxley model of a neuron using the Message Passing Interface (MPI) Application Program Interface (API). MPI was used to parallelize the program to run using cluster computing to test the execution time in comparison to running the code using traditional sequential computing. A sequential implementation of the model was modified to run in parallel utilizing MPI. This parallel program was used to test the execution time when modifying certain model parameters and also to test the load balancing that occurs between the different processors running the program. In order to evaluate the performance of the parallel computer, the parallel times were compared to the sequential times using the speedup between the times. The parallel times of every trial had a believable speedup compared to the sequential times, which means that the experiment was a success.

Design Methodology

The Hodgkin-Huxley neuron model describes how the electrical signals in any given neuron form and propagate throughout the neuron using mathematical principles such as differential equations. The model takes several parameters of the neuron such as the amount of dendrites and length of the dendrites to describe how the neuron will behave under the circumstances. The model is relatively complex considering the use of several coupled differential equations; this results in difficulty for even modern processors to emulate the model when utilizing traditional sequential computing using large input parameters.

One way to deal with the computational complexity of the model is to run the program using several processors instead of just one, allowing multiple computations to occur concurrently. Many parts of the computation involved in this model happen to be easily separable, meaning that different parts of the code can be sent to different processors without requiring heavy waiting for other processors to finish their computations and communicate the results.

The MPI API is a commonly used tool which allows a programmer to write C code which can run a multiple processor system in parallel. MPI uses a master-slave relationship where one master node communicates with several slave nodes that perform the computations. MPI incorporates several functions alongside normal C code that handle most communications between the different processors allowing for minimal work needed from the programmer.

The ease of use obtained from MPI makes the parallelization of the model relatively trivial. The code that is responsible for the calculations can remain mostly the same because the model did not change nor did the fundamentals of C code, the only thing that changed was the addition of MPI which runs the code differently. The only thing that needs to be done to the code is to add in MPI functions to appropriately separate the work load, making sure to use every function according to both the MPI documentation and the initial code structure to avoid any errors in the end result. The code can be run using a simple command in a cluster computing environment which can input both the model inputs and the desired number of processes.

Results and Analysis

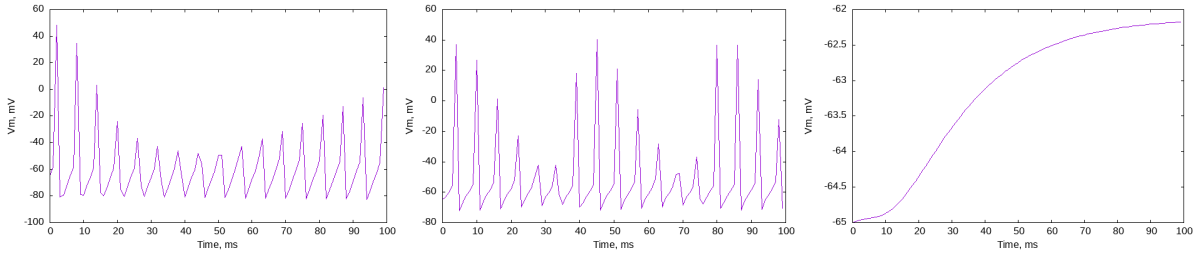
Table 1 compares the execution time of several program runs with 15 dendrites and a varying number of compartments per dendrite. For each compartment setting, the performance is compared for 0 (sequential), 5, and 12 slave processes. The 10-compartment and 100-compartment runs both have the best execution time and speedup with 5 slave processes. With 12 slave processes, the efficiency decreases for each of these workloads due to the additional communication time required to accumulate the results of computations. However, in the 1000-compartment case, the run with 12 slave processes fares best because each process completes enough computational work to outweigh the cost of communication. In general, the speedup is not the same as the number of slave processes because there is a significant amount of parallelization overhead from communication and synchronization operations – the ideal speedup cannot be achieved.

Number of Slaves	Dendrites	Compartments	Exec Time (sec)	Speedup
0, sequential	15	10	31.7	1
5 (srun –n 6)	15	10	7.8	4.1
12 (srun –n 13)	15	10	17.5	1.8
0, sequential	15	100	198.3	1
5 (srun –n 6)	15	100	41.1	4.8
12 (srun –n 13)	15	100	49.3	4.0
0, sequential	15	1000	1862.2	1
5 (srun –n 6)	15	1000	394.7	4.7
12 (srun –n 13)	15	1000	353.3	5.3

Table 1: Effect of Dendrite Length on Computational Load

Fig. 1 compares the soma membrane potential for three different experiments from Table 1, where the dendrite length is varied. Fig. 1a shows a falling and rising spiking pattern in the membrane potential. This pattern is compressed as a result of the longer dendrite in Fig. 1b, with three iterations of it visible on the same time scale. Additionally, the pattern appears over a more narrow voltage range in Fig. 1b. Finally, in Fig. 1c with the longest dendrite, the voltage range is extremely narrow (only 3 mV) and only a gradual increase in the soma membrane potential is captured.

Table 2 shows a comparison of execution time for a constant 10 compartments, but varying the number of dendrites. Note that the first three rows of Table 2 are repeated from Table 1. By increasing the number of dendrites to 150 and 1500, the benefits of parallelization become more dramatic because each process has more computational work in between communications, lowering the communication-to-computation ratio. In the last row of Table 2, 1500 dendrites with 12 processes, the speedup is the highest because 1500 is divisible by 12, meaning there is also perfect load balancing among the slave processes. The speedup from increasing the number of dendrites by a factor of 10 is higher in Table 2 than for increasing the dendrite length by the same factor in Table 1. However, the actual execution time is affected more by increasing the number of dendrites.



(a) 10 compartments; range -100 mV to 60 mV.

(b) 100 compartments; range -80 mV to 60 mV.

(c) 1000 compartments; range -65 mV to -62 mV.

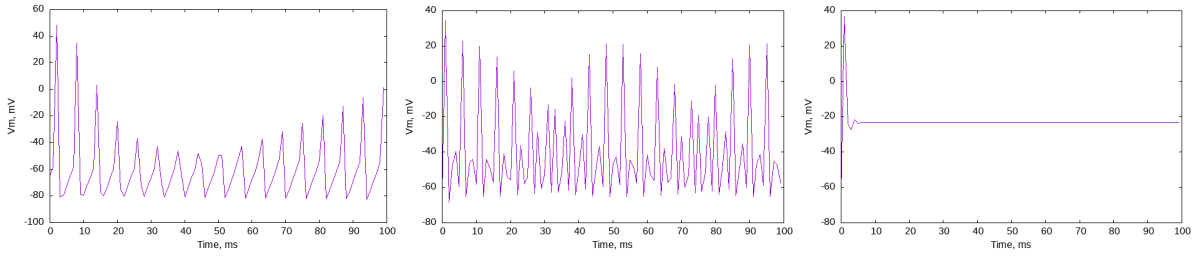
Figure 1: Membrane potential (mV) over 100 ms with 1 million integration steps using 15 dendrites and 12 slave processes. The dendrite length used to produce each plot increases from left to right.

Number of Slaves	Dendrites	Compartments	Exec Time (sec)	Speedup
0, sequential	15	10	31.7	1
5 (srun –n 6)	15	10	7.8	4.1
12 (srun –n 13)	15	10	17.5	1.8
0, sequential	150	10	310.0	1
5 (srun –n 6)	150	10	57.0	5.4
12 (srun –n 13)	150	10	47.8	6.5
0, sequential	1500	10	3278.6	1
5 (srun –n 6)	1500	10	564.3	5.8
12 (srun –n 13)	1500	10	327.9	10.0

Table 2: Effect of Number of Dendrites on Computational Load

Fig. 2 displays a similar comparison to the membrane potentials shown in Fig. 1, but for increasing number of dendrites. Fig. 2a is identical to Fig. 1a. In Fig. 2b, it is shown that increasing the number of dendrites results in more frequent oscillations in the membrane potential, in addition to a more compressed overall pattern in comparison to Fig. 2a. In Fig. 2c, with the most dendrites, a single dramatic spike is present at the beginning of the simulation, with no activity after about 5 ms. This is in direct contrast to Fig. 1c, which showed a very gradual and small potential increase.

A simple test for load imbalance with 10 slave processes was conducted, and the results are shown in Table 3. Experiment #1 has perfect load balancing among the slave processes – 30 dendrites divided among 10 processes. In experiment #2, there are more than 30 dendrites, so the execution time is naturally higher because some of the slave processes have to do more work than in experiment #1. There is also worse load balancing than in experiment #1 because 33 is not divisible by 10. In experiment #3, the execution time is very similar to experiment #2 despite the fact that there are more dendrites. This is because in experiment #2, there are three slave processes which have more work (exactly one more dendrite) than all the others, and in experiment #3 there are six processes affected in that way, but their work is still done concurrently. These results demonstrate effective load balancing among the slave processes.



(a) 15 dendrites; range -100 mV to 60 mV.

(b) 150 dendrites; range -80 mV to 40 mV.

(c) 1500 dendrites; range -80 mV to 40 mV.

Figure 2: Membrane potential (mV) over 100 ms with 1 million integration steps using 10 compartments per dendrite and 12 slave processes. The number of dendrites used to produce each plot increases from left to right.

#	Number of Slaves	Dendrites	Compartments	Exec Time (sec)
1	10 (srun –n 11)	30	100	60.2
2	10 (srun –n 11)	33	100	76.3
3	10 (srun –n 11)	36	100	77.3

Table 3: Test for Load Imbalance

The results of additional load imbalance testing are compiled in Table 4. Experiments #1-3 show the effects of load imbalance when there is a high dendrite length and few dendrites, and experiments #4-6 show load imbalance for many shorter dendrites. Experiments #1 and #5 exhibit the best load balancing because the number of dendrites is divisible by the number of slave processes. The 6-dendrite case has 194.7% of the execution time of the 5-dendrite case due to the single slave process which has to handle an additional 1000-compartment dendrite. In comparison, the 1501-dendrite experiment has 103.6% of the 1500-dendrite execution time, also due to a single slave process handling one extra (shorter) dendrite. So, the load imbalance is much more dramatic in the case of the few long dendrites than for the many short ones. To improve this behavior, specifically the worst case load imbalance demonstrated in experiment #2, the parallel program might be modified to allow several slave processes to handle the compartments in a single dendrite after all dendrites have been accounted for.

#	Number of Slaves	Dendrites	Compartments	Exec Time (sec)
1	5 (srun –n 6)	5	1000	137.0
2	5 (srun –n 6)	6	1000	266.8
3	5 (srun –n 6)	7	1000	261.5
4	5 (srun –n 6)	1499	10	563.1
5	5 (srun –n 6)	1500	10	564.3
6	5 (srun –n 6)	1501	10	584.5

Table 4: Test for Load Imbalance

Conclusion

This work demonstrates the challenges and feasibility of implementing Hodgkin-Huxley neuron simulation as a parallel program. Single-program, multiple-data parallelism can be exploited by having several slave processes each handle a subset of the dendrites used in the neuron model. The appropriate number of processes varies depending on the computational workload and must be chosen carefully to avoid unnecessary communication overhead. The load balancing of this parallel program is effective, but could be improved by exploiting finer-grain parallelism that exists in the compartment calculations for each individual dendrite. Overall, this exercise was successful in demonstrating the value of parallel programming in the simulated neuron application.