Multiprocessor Systems – Assignment 1

Abstract

This project includes the parallel implementation of a simple compartmental Hodgkin-Huxley neuron model, along with the given sequential implementation. The Message Passing interface (MPI) is used in the parallel implementation of the model, for communication between processes. Both sequential and parallel implementations are run various configurations, i.e. the number of compartments in a dendrite and the number of dendrites, and the results are recorded and tabulated along with parallel vs sequential speedup.

Design Methodology

Parallel implementation of a simple compartmental Hodgkin-Huxley neuron model is developed using Message Passing Interface (MPI). Basically MPI provides a parallel programming model, using which processes communicate with each other. In the sequential code, calculation of injected current into soma from each dendrite is implemented sequentially. Since this operation is independent in each dendrite, it can be parallelized. Hence this operation is parallelized by dividing the work evenly among given number of processes. This division of work among processes is implemented using rank of each process. Each process calculates the current injected into soma by one dendrite, per iteration. If the number of dendrites is greater than the number of processes, then each process calculates and accumulates the current generated by multiple dendrites in multiple iterations. The total work is evenly distributed among processes using the rank of each process. The accumulated current from multiple dendrites by each slave process is sent to master process using MPI_Send function and received at master process using MPI_Receive function. Master process accumulates the current generated by all dendrites and uses the accumulated current to update the potential of soma for each integration step. After each integration step, the updated y[0] value which is used to calculate current injected by each dendrite into soma, is sent from master process to all slave processes using MPI_Send and MPI_Receive functions. Master process records soma membrane potential for each ms. Master process creates and updates the data file and generates a plot of membrane potential of soma against time. Unique message tags are used for all MPI sends and receives in each step.

Results and Analysis

Number of Slaves	Dendrites	Compartments	Exec Time (sec)	Speedup
0, sequential	15	10	82.037273	1
5 (mpirun –np 6)	15	10	41.187848	1.991783426
12 (mpirun –np 13)	15	10	284.155174	0.288705892
0, sequential	15	100	534.833039	1
5 (mpirun –np 6)	15	100	258.97596	2.065184116
12 (mpirun –np 13)	15	100	206.451487	2.590599113
0, sequential	15	1000	5034.31304	1
5 (mpirun –np 6)	15	1000	2440.444019	2.062867659
12 (mpirun –np 13)	15	1000	1670.380262	3.013872442

Table 1. Effect of Dendrite Length on Computational Load

From the results of parallel implementation, it can be observed that execution time of parallel implementation is lesser than that of sequential execution, in all cases except in the third configuration in the above table. This is because 6 or 13 processes run in parallel and each process work on a dendrite, which makes execution faster. Also the speedup increases with increase in dendrite length, i.e. number of compartments. But the speedup is not the same as number of parallel processes, because of communication overhead, i.e. communication between parallel processes using MPI. Spiking patterns are observed as the membrane potential of soma increases as current flows into from all dendrites into soma and then once membrane potential reaches the threshold value, soma discharges current and then charges again after a certain point. This process is recursive. So, if dendrite length increases the current injected by each dendrite into soma decreases as energy dissipates while passing through more number of compartments (resistance).

Number of Slaves	Dendrites	Compartments	Exec Time (sec)	Speedup
0, sequential	15	10	82.037273	1
5 (mpirun –np 6)	15	10	41.187848	1.991783426
12 (mpirun –np 13)	15	10	284.155174	0.288705892
0, sequential	150	10	812.576769	1
5 (mpirun –np 6)	150	10	285.93207	2.841852504
12 (mpirun –np 13)	150	10	451.127092	1.801214743
0, sequential	1500	10	8178.429983	1
5 (mpirun –np 6)	1500	10	2745.955969	2.978354378
12 (mpirun –np 13)	1500	10	1305.927607	6.262544676

Table 2. Effect of Number of Dendrites on Computational Load

From the above results, it can be inferred that the execution time of parallel implementation is lesser than that of sequential execution, except in on case. This is expected because 6 or 13 processes run in

parallel, each working on a dendrite at a time. Also the speedup increases with increase in number of dendrites. Compared to data in table 1, speedup increases more with increase in number of dendrites than with increase in number of compartments. This is because computation done on each dendrite is parallel, but computation done on each compartment in a dendrite is still sequential. With increase in number of dendrites, the total current injected into soma for each step increases as more number of dendrites contribute to the total current injected into soma. Hence frequency of spikes increases with increase in number of dendrites.

#	Number of Slaves	Dendrites	Compartments	Exec Time (sec)
1	10 (mpirun –np 11)	30	100	269.035642
2	10 (mpirun –np 11)	33	100	265.624171
3	10 (mpirun –np 11)	36	100	348.590714

Table 3. Test for Load Imbalance

Experiment #1 execution time is similar to experiment #2 execution time, because in both cases all or some processes execute for three iterations to work on all dendrites. But experiment #3 took longer to execute, because in this case some processes run for 4 iterations to work on all dendrites. From all three experiments in the above table, it can be inferred that if the number of dendrites is a multiple of total number of processes, then the computational load is evenly balanced among all processes. But if the number of dendrites is not a multiple of total number of processes, few processes remain idle in the final iteration which causes load imbalance in the final iteration. But still the computational load is evenly balanced in the remaining iterations. Hence it can be said that the program is effective at balancing computational work load.

#	Number of Slaves	Dendrites	Compartments	Exec Time (sec)
1	5 (mpirun –np 6)	5	1000	828.981505
2	5 (mpirun –np 6)	6	1000	821.435696
3	5 (mpirun –np 6)	7	1000	1576.263882
4	5 (mpirun –np 6)	1199	10	2194.558018
5	5 (mpirun –np 6)	1200	10	2202.096134
6	5 (mpirun –np 6)	1201	10	2199.618523

Table 4. Effect of Load Imbalance

Experiment #3 with 7 dendrites running on 6 processes is the worst case, because only one process works on a dendrite in the 2nd iteration and the remaining process remains idle in the second iteration. Hence the execution time in the experiment #3 is almost twice as that of experiment #1 & #2. Experiment #4 with 1199 dendrites and 10 compartments per dendrite running on 6 processes is much more effective than Experiment #3, because computational load is evenly balanced among all processes

for all 200 iterations except the last iteration. We can improve the performance in experiment #3 by increasing the number of slaves by 1, as work on all 7 dendrites would be completed in one iteration and the load would be evenly balanced among all processes.

Conclusion

This project helps to get familiar with parallel programming and message passing interface (MPI). From the results it can be inferred that parallel vs sequential speedup increases with increase in problem size, i.e. increase in number of dendrites and/or number of compartments. Also we can observe different spike patterns with different configurations, i.e. different number of dendrites and/or different number of compartments.