Introduction of High-Performance Computing for Neuroinformatics

Yamazaki Tadashi*

The University of Electro-Communications

November 21, 2017

Abstract

Neural network simualtion would spend to much computational time as the network size increases and the detail becomes more realitstic. In this tutorial, first we learn the basics of neural network simulation, and then learn methods to accelerate simulation by parallel computing technology. This tutorial is not just a lecture course but with a hands-on where we write the code and test on our cluster machine.

Contents

* Email: aini17@numericalbrain.org, Webpage: http://numericalbrain.org/en/

1 Introduction

A rough schedule is as follows:

13:00–13:10	Opening
13:10–13:30	Log into our cluster machine
13:30–14:30	First course: Introduction to neural network simulation
14:30–14:40	Break
14:40–15:40	Second course: Parallel simulation by OpenMP
15:40–15:50	Break
15:50–16:50	Third course: Parallel simulation by MPI and hybrid parallelization
16:50–17:00	Closing

This tutorial is supported by MEXT Post-K Exploratory Research #4 "Understanding the neural mechanisms of thoughts and its applications to AI": (a) Bottom-up whole insect brain simulation to understand the elemental intelligence of insects, and (b) Big Brain-Data Analysis, Whole Brain-Scale Simulation, and Brain-Style AI Architecture[*1].

2 Opening

Let's introduce yourself.

3 Log into our cluster machine

I will give your username on our cluster in exchange for your ssh public key. If you don't have your ssh key-pair, create it as follows:

```
[yourname@yourmachine ~]$ ssh-keygen -t rsa
```

You are prompted to input your passphrase, and after that, you will have `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub`. Give `id_rsa.pub` to me.

Log into our cluster machine. The machine name is `plato.sim.neuroinf.jp`.

```
[yourname@yourmachine ~]$ ssh plato.sim.neuroinf.jp -l <username>
```

wehere `<username>` is the given username on the machine.

[*1] `https://brain-hpc.jp/`

4 First course: Introduction to neural network simulation

4.1 Simulation of one neuron

Let's start with simulation of just one neuron[3]. The most representative neuron model would be the Hodgkin-Huxley model, but in this tutorial, we employ much simpler leaky integrate-and-fire model (LIF).

The current-based LIF model is described as follows:

$$\tau \frac{dv}{dt} = -\left(v(t) - V_{\text{leak}}\right) + RI_{\text{ext}}(t), \tag{1}$$

$$v(t) > \theta \Rightarrow \text{Spike (t)} = 1, v(t) \leftarrow, V_{\text{reset}}, \tag{2}$$

$$v(0) = V_{\text{init}}, \tag{3}$$

where $v(t)$ (mV) is the membrane potential at time t, τ (ms) is the time constant, V_{leak} (mV) is the resting potential, R (MΩ) is the membrane resistance, $I_{\text{ext}}(t)$ (nA) is the external current injected at t, θ (mV) is the threshold for spike generation, V_{reset} (mV) is the reset potential, and V_{init} (mV) is the initial value of the membrane potential.

Equation (1) describes the basic dynamics of the membrane potential. With V_{leak} as the equilibrium value, the membrane potential value asymptotes $RI_{\text{ext}}(t)$ with time constant τ. Equation. (2) describes the condition of the spike generation. When the membrane potential exceeds the threshold, we regard the neuron makes the spike at the time (Spike (t) = 1), and reset the membrane potential. Equation (3) is the initial value of the membrane potential.

To solve the differential equation by a computer numericallly, we convert it to a difference equation. We consider a sufficiently short time interval Δt, and approximate

$$\frac{dv}{dt} \approx \frac{\Delta v(t)}{\Delta t}. \tag{4}$$

On the other hand, we consider the taylor expansion of $v(t)$ with Δt as

$$v(t + \Delta t) = v(t) + \frac{dv}{dt}\Delta t + \frac{1}{2!}\frac{d^2v}{dt^2}\Delta t^2 + \cdots, \tag{5}$$

and ignore the $O(\Delta t^2)$ and smaller terms under the assumption of sufficiently small Δt value. We have

$$v(t + \Delta t) \approx v(t) + \frac{dv}{dt}\Delta t. \tag{6}$$

Finally, these two equations yield

$$v(t + \Delta t) \approx v(t) + \Delta v(t), \tag{7}$$

where

$$\Delta v(t) = \frac{\Delta t}{\tau}\left(-\left(v(t) - V_{\text{leak}}\right) + I_{\text{ext}}\right). \tag{8}$$

Once the initial value $v(0)$ is give, we can obtain the value of $v(t)$ approximately at $t = 0, \Delta t, 2\Delta t, \cdots$ by solving Eq. (7) sequentially. This numerial method is called **the explicit Euler method**.

Now, let's try the following code:

Listing 1 `lif.c`

```
1 #include<stdio.h>
2
3 #define TAU 20.0
4 #define V_LEAK -65.0
5 #define V_INIT (V_LEAK)
6 #define V_RESET (V_LEAK)
```

```
7  #define THETA -55.0
8  #define R_M 1.0
9  #define DT 1.0
10 #define T 1000.0
11 #define NT 1000 // ( T / DT )
12 #define I_EXT 12.0
13
14 void loop ( void )
15 {
16   double v = V_INIT;
17   for ( int nt = 0; nt < NT; nt++ ) {
18     printf ( "%f␣%f\n", DT * nt, v );
19     double i_ext = ( DT * nt < 100.0 || 900 < DT * nt) ? 0 : I_EXT;
20     double dv = ( DT / TAU ) * ( - ( v - V_LEAK ) + R_M * i_ext );
21     v += dv;
22     if ( v > THETA ) {
23       printf ( "%f␣0\n", DT * nt ); // print spike with membrane potential = 0 mV
24       v = V_RESET;
25     }
26   }
27 }
28
29 int main ( void )
30 {
31   loop ( );
32
33   return 0;
34 }
```

This code simulates the neuron dynamics for 1000 msec (= 1 sec) with temporal resolution of $\Delta t = 1$. We inject the external current of $I_{\rm ext} = 12$ nA during 100 msec to 900 msec. We set the membrane resistance at 1 MΩ for simplicity. The code calculates the value of $v(t)$ for each Δt sequentially with the initial value of $v(0) = -65$ mV.

The code starts with `main` in line 29, and it just calls function `loop` (line 31). The function `loop` (lines 14–27) is the core of the simulation, so let's take a look closely.

Line 16 defines the value of the membrane potential `v`, and set the initial value at `V_INIT`= −65 mV, which is defined in line 5.

Line 17 is the loop with respect to time. We set the duration of the simulation at $T = 1000$ msec (line 10), and the time is update with temporal resolution of $\Delta t =$ `DT` $= 1$ msec (line 9). So, the number of iterations `NT` is `NT` $= T/DT = 1000$. We count the iteration with variable `nt`.

Line 18 prints the value of $v(t)$ with the current time.

Line 19 sets the amplitude of the external current. We use a ternary operator to set the amplitude at `i_ext`=`I_EXT` during 100–900 msec, and 0 otherwise. `I_EXT` is defined in line 12.

Line 20 calculated $\Delta v(t)$ according to Eq. (8). τ is set at `TAU` = 20 ms in line 3, and $R_M =$ `R_M` $= 1$ MΩ is defined in line 8.

Line 21 updates $v(t)$ according to Eq. (7).

Lines 22–25 calculates the spike generation. If $v(t)$ exceeds the threashold `THETA` (line 22), we print the membrane potential of 0 mV with the current time (line 23), and reset $v(t)$ to `V_RESET`. `THETA` $= -55$ mV is defined in line 7, `V_RESET` $= -65$ mV in line 6.

Let's compile the code and try. Compile the code as follows:

```
[tyam@plato tutorial]$ gcc -std=c99 -Wall -O3 -o lif lif.c
```

where `-std=c99` option tells the compiler to follow C99 specification, `-Wall` option prints all warnings and is strongly recommmended, and `-O3` option performs the most extreme code optimization. When succeeded, the executable file `lif` is generated. Try it as

```
[tyam@plato tutorial]$ ./lif
```

You will see the flow of numbers, which represents the time and the value of $v(t)$ at time. To see what happens, redirect the output to the file:

```
[tyam@plato tutorial]$ ./lif > lif.dat
```

and plot the result with `gnuplot`.

```
[tyam@plato tutorial]$ gnuplot
       G N U P L O T
(...snip...)
Terminal type set to 'x11'
gnuplot> plot 'lif.dat' with line
```

Then, you will obtain the plot of the membrane potential as in Fig. 1. We can confirm that the neuron makes spikes regularly during 100–900 msec.

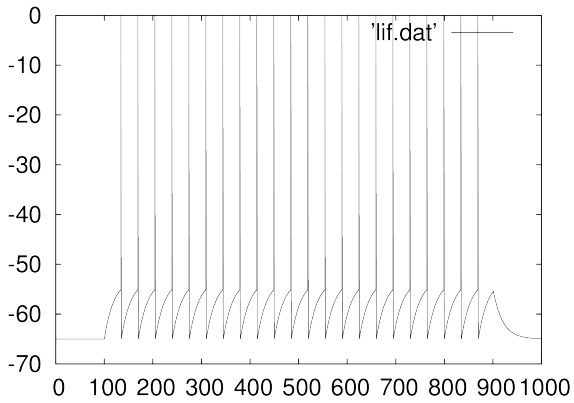

Fig. 1 Simulated activity of one neuron. The horizontal axis represents time (msec), and the vertical axis the membrane potential (mV).

Caution! Be careful with the unit of parameters. If you regard the unit of the time as sec but not msec, and set $T = 1.0$, $\Delta t = 0.001$, then the simulation result becomes incorrect. Always use either Physiologial Unit as this code or SI Unit.

4.2 Simulation of two neurons

Because the simplest neural network is composed of 2 neurons, let's create it.

When the neurons are not connected synaptically and so the are completely independent, the code will become almost the same with `lif.c`. In particular, we let `v` and `dv` be arrays `v[2]` and `dv[2]`, respectively, and set one of the initial values of `v` 10 mV lower. The resulting code becomes as follows:

Listing 2 `lif2.c`

```
1 #include<stdio.h>
2
3 #define TAU 20.0
4 #define V_LEAK -65.0
5 #define V_INIT (V_LEAK)
```

```
6  #define V_RESET (V_LEAK)
7  #define THETA -55.0
8  #define R_M 1.0
9  #define DT 1.0
10 #define T 1000.0
11 #define NT 1000 // ( T / DT )
12 #define I_EXT 12.0
13
14 void loop ( void )
15 {
16   double v [ 2 ] = { V_INIT, V_INIT - 10.0 };
17   double i_ext = I_EXT;
18   for ( int nt = 0; nt < NT; nt++ ) {
19     printf ( "%f␣%f␣%f\n", DT * nt, v [ 0 ], v [ 1 ]);
20     double dv [ 2 ];
21     dv [ 0 ] = ( DT / TAU ) * ( - ( v [ 0 ] - V_LEAK ) + R_M * i_ext );
22     dv [ 1 ] = ( DT / TAU ) * ( - ( v [ 1 ] - V_LEAK ) + R_M * i_ext );
23     v [ 0 ] += dv [ 0 ];
24     v [ 1 ] += dv [ 1 ];
25     if ( v [ 0 ] > THETA ) {
26       printf ( "%f␣0␣%f\n", DT * nt, v [ 1 ] ); // print spike with membrane potential = 0 mV
27       v [ 0 ] = V_RESET;
28     }
29     if ( v [ 1 ] > THETA ) {
30       printf ( "%f␣%f␣0\n", DT * nt, v [ 0 ] ); // print spike with membrane potential = 0 mV
31       v [ 1 ] = V_RESET;
32     }
33   }
34 }
35
36 int main ( void )
37 {
38   loop ( );
39
40   return 0;
41 }
```

The differences are as follows. We let `v` an array and set the initial value appropriately (line 16). We injected the external current from $t = 0$ msec (line 17). We also let `dv` an array (line 20). We calculated `dv` and `v` twice while incrementing the index (lines 21–24). We changed the output format (lines 26 and 30).

We should do the calculation of the two neurons using `for` loop, but we did not for clarity.

Let's compile the code and plot the result.

```
[tyam@plato tutorial]$ gcc -std=c99 -Wall -O3 -o lif2 lif2.c
[tyam@plato tutorial]$ ./lif2 > lif2.dat
[tyam@plato tutorial]$ gnuplot
       G N U P L O T
(...snip...)
Terminal type set to 'x11'
gnuplot> plot 'lif2.dat' using 1:2 with line, 'lif2.dat' using 1:3 with line
```

You will have the plot of the membrane potential as in Fig. 2. Because the initial condition is different, the spike timings of the neurons are shifted, while the waveforms are identical.

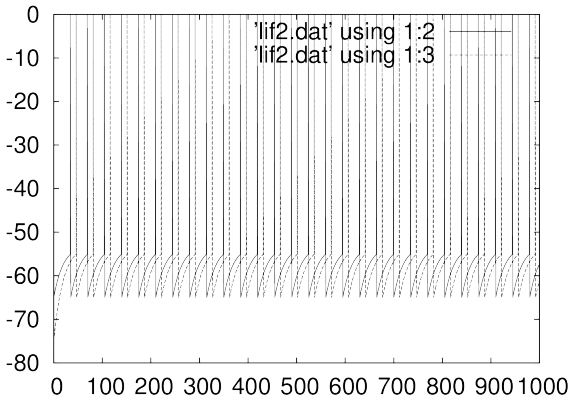

Fig. 2 Dynamis of the two independent neurons. Conventions as in Fig. 1.

4.3 Simulation of a network with two mutually connected neurons

Finally, let's build a true neural network by connecting the two neurons synaptically. Here, we employ exponential synapses, the simplest form of the synaptic dynamics:

$$\tau_{\mathrm{syn}} \frac{dg_i}{dt} = -g_i(t) + w \cdot \mathrm{Spike}_{(i+1)\%2}(t), \tag{9}$$

where i is the neuron index($i \in \{0, 1\}$), τ_{syn} is the time constant, $g_i(t)$ is the postsynaptic potential, w is the synaptic weight, and $\mathrm{Spike}_{(i+1)\%2}(t)$ represents the spike of the other neuron (0 or 1)[*2]

We derive the difference version of the equation, and solve it by the explicit Euler method as follows:

$$g_i(t + \Delta t) = g_i(t) + \frac{\Delta t}{\tau_{\mathrm{syn}}} \left(-g_i(t) + w \cdot \mathrm{Spike}_{(i+1)\%2}(t) \right), \tag{10}$$

where the initial value $g_i(0)$ is set at 0 mV. By the way, do not forget to add the term $g(t)$ to the right hand side of the equation of the membrane potential.

Listing 3 `lif2net.c`

```
1  #include<stdio.h>
2
3  #define TAU 20.0
4  #define V_LEAK -65.0
5  #define V_INIT (V_LEAK)
6  #define V_RESET (V_LEAK)
7  #define THETA -55.0
8  #define R_M 1.0
9  #define DT 1.0
10 #define T 1000.0
11 #define NT 1000 // ( T / DT )
12 #define I_EXT 12.0
13 #define TAU_SYN 5.0
14 #define W 10.0 //-10.0
15
16 void loop ( void )
17 {
18   double v [ 2 ] = { V_INIT, V_INIT - 10. };
```

[*2] Notice that $(i + 1)\%2 = 1$ when $i = 0$, and 0 when $i = 1$.

```
19   double i_ext = I_EXT;
20   double g [ 2 ] = { 0., 0. };
21   int spike [ 2 ] = { 0, 0 };
22   for ( int nt = 0; nt < NT; nt++ ) {
23     printf ( "%f␣%f␣%f\n", DT * nt, v [ 0 ], v [ 1 ]);
24     double dv [ 2 ];
25     dv [ 0 ] = ( DT / TAU ) * ( - ( v [ 0 ] - V_LEAK ) + g [ 0 ] + R_M * i_ext );
26     dv [ 1 ] = ( DT / TAU ) * ( - ( v [ 1 ] - V_LEAK ) + g [ 1 ] + R_M * i_ext );
27     double dg [ 2 ];
28     dg [ 0 ] = ( DT / TAU_SYN ) * ( - g [ 0 ] + W * spike [ 1 ] );
29     dg [ 1 ] = ( DT / TAU_SYN ) * ( - g [ 1 ] + W * spike [ 0 ] );
30     v [ 0 ] += dv [ 0 ];
31     v [ 1 ] += dv [ 1 ];
32     g [ 0 ] += dg [ 0 ];
33     g [ 1 ] += dg [ 1 ];
34     spike [ 0 ] = ( v [ 0 ] > THETA );
35     if ( v [ 0 ] > THETA ) {
36       printf ( "%f␣0␣%f\n", DT * nt, v [ 1 ] ); // print spike with membrane potential = 0 mV
37       v [ 0 ] = V_RESET;
38     }
39     spike [ 1 ] = ( v [ 1 ] > THETA );
40     if ( v [ 1 ] > THETA ) {
41       printf ( "%f␣%f␣0\n", DT * nt, v [ 0 ] ); // print spike with membrane potential = 0 mV
42       v [ 1 ] = V_RESET;
43     }
44   }
45 }
46
47 int main ( void )
48 {
49   loop ( );
50
51   return 0;
52 }
```

We changed the code as follows: First, we defined the variable for synapses `g` (line 20), and spikes `spike` (line 21). The synaptic inputs are added to the membrane potentials (lines 25, 26). Lines 28, 29 calculates the synaptic inputs, and lines 32, 33 updates the values. Lines 34, 39 generates the spikes.

Let's compile the code, try, and plot the results.

```
[tyam@plato tutorial]$ gcc -std=c99 -Wall -O3 -o lif2net lif2net.c
[tyam@plato tutorial]$ ./lif2net > lif2net.dat
[tyam@plato tutorial]$ gnuplot
       G N U P L O T
(...snip...)
Terminal type set to 'x11'
gnuplot> plot 'lif2net.dat' using 1:2 with line, 'lif2net.dat' using 1:3 with line
```

You will have the plot of the membrane potential as in Fig. 3. You can see that the spike timing is grdually synchronized with time.

Exercise 1.

See what happens when the neurons are connected in inhibitory manner. You can test this by setting the sign of `W` value negative.

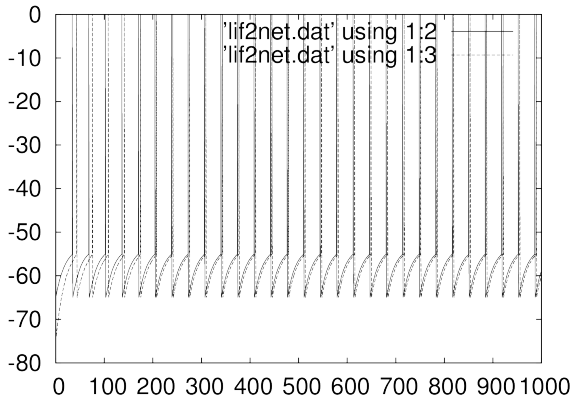

Fig. 3 Neural network dynamics composed of two mutually connected neurons. Conventions as in Fig. 1.

Exercise 2.
Examine why this happens.

That's all for the first course. Break!

5 Second course: Parallel simulation by OpenMP

5.1 Simulation of a random network

The network model with two neurons that we considered in the first course is so small that the calculation completes almost immediately. We extend the network size much larger. In particular, we consider a random network model composed of 4000 neurons, where 80% of them are excitatory while the rest are inhibitory. They are connected randomly with probability $p = 0.02$[4]. The network is a standard model for the benchmark of general neural network model simulators [5]。

The equation of the membrane potential is identical to Eqs. (1)–(3):

$$\begin{aligned}
\tau \frac{dv}{dt} &= -\left(v(t) - V_{\text{leak}}\right) + ge(t) + gi(t), \\
v(t) > \theta &\Rightarrow \text{Spike (t)} = 1, v(t) \leftarrow V_{\text{reset}}, \\
v(0) &= V_{\text{init}},
\end{aligned}$$

where $v(t)$ is the membrane potential at time t, $\tau = 20$ ms is the time constant, $V_{\text{leak}} = -49$ mV is the resting potential, $ge(t)$, $gi(t)$ are respectively excitatory and inhibitory postsynaptic potentials, $\theta = -50$ mV is the threshold for spike generation, $V_{\text{reset}} = -60$ mV is the reset potential, $V_{\text{init}} = -60 + 10 \times \text{rand}(t)$ is the initial value of the membrane potential, and rand(t)is the uniform random number on $[0, 1)$. On the other hand, we calculate the postsynaptic potential as follows:

$$\begin{aligned}
\tau_e \frac{dge}{dt} &= -ge(t) + \sum_{j \in \text{Exc}} w_e \cdot \text{Spike}_j(t), \\
\tau_i \frac{dgi}{dt} &= -gi(t) + \sum_{j \in \text{Inh}} w_i \cdot \text{Spike}_j(t),
\end{aligned} \tag{11}$$

where $\tau_e, \tau_i = 5, 10$ ms are respectively the time constants, Exc, Inh are respectively excitatory and inhibitory neuron groups, $w_e, w_i = +1.62, -9$ mV are respectively the changes of postsynaptic potentials by one spike input, and $\text{Spike}_j(t) \in \{0, 1\}$ represents 1 when neuron j makes a spike at time t and 0 otherwise.

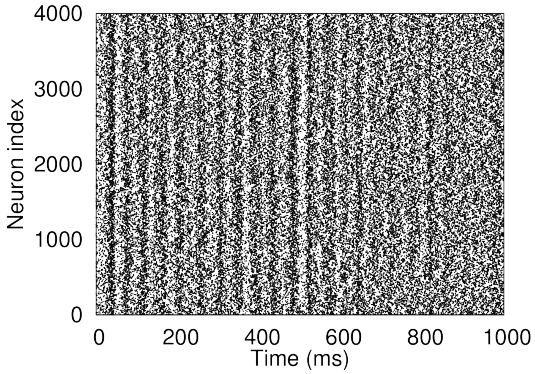

Fig. 4 Raster plot of the random network model. The horizontal axis is time (msec) and the vertical axis is the neuron index. Each dot represents one spike.

The code is in Appendix. The code generates the file `spike.dat`. When we plot the file, we will obtain the raster plot as in Fig. 4. We no longer plot the membrane potentials, because it is extremely difficult to see the waveforms for 4000 neurons. Instead, we plot only the spike times as the raster plot.

The code in general looks like this:

Listing 4 `randomnet.c`

```
1  void loop ( void )
2  {
3    for ( int nt = 0; nt < NT; nt++ ) {
4      for ( int i = 0; i < N; i++ ) {
5        calculateSynapse ( i );
6        updateMembranePotential ( i );
7      }
8      outputSpike ( nt );
9    }
10 }
```

The loop for time (line 3) is the same as it was. Now, we have much more neurons, so we calculate the neurons using `for` loop (line 4). In the loop, first we calculate the synaptic inputs (line 5), and then update the membrane potential (line 6). Once all the calculations are done, we output the spike information to the file (line 8). The internals of functions `calculateSynapse` and `updateMembranePotential` are something that you expect.

Our cluster machine spends 25 secs for one trial of the simulation of the network model. Please try.

5.2 Acceleration of simulation by Parallel computing

As the numbers of neurons and synapses increse, the computational time for simulation becomes very long. A potential way to solve this issue is parallel computing using the multi-core functioning of a single CPU or multi CPUs through dedicated libraries, and dedicated hardware called accelerators such as graphics processing units (GPUs). Generally speaking, we split the loop for neurons (line 4 and its internal), and compute groups of neurons independently in parallel.

5.3 OpenMP

OpenMP is a standard specification to use multi cores in a single CPU in parallel [7]. The C compiler in GNU Compiler Collection (GCC) has this function in default. To use OpenMP, we include `#include <omp.h>` in the header area of the source code, and compile the code with `-fopenmp` option.

It is extremely easy to parallelize the loop with respect to neurons with OpenMP. We just add a single line as follows:

Listing 5 `omp.c`

```
1  void loop ( void )
2  {
3    for ( int nt = 0; nt < NT; nt++ ) {
4  #pragma omp parallel for
5      for ( int i = 0; i < N; i++ ) {
6        calculateSynapse ( i );
7        updateMembranePotential ( i );
8      }
9      outputSpike ( nt );
10   }
11 }
```

`#pragma omp parallel for` in line 4 is a directive of OpenMP. By this directive, the `for` loop in the next line, the calculations of synapses and membrane potentials for neuonrs, is split and executed in parallel. Note that the internals of the `for` loop can be calculated independently.

The schematic of the parallelization is as in Fig. 5. In the case of 1 CPU (or 1 core), 1 thread (i.e., computational unit) calculates neurons one by one sequentially (Fig. 5A). When using OpenMP, multiple cores in a CPU are used simultaneously, and multiple threads perform neuron calculatation independently (Fig. 5B). So, theoretically, the computational time is shorten by the number of involved threads.

When we do the calculation using all 4 cores [*3], the computational time is shorten to 12.5 sec, resulting in $25/12.5 = 2$ time acceleration.

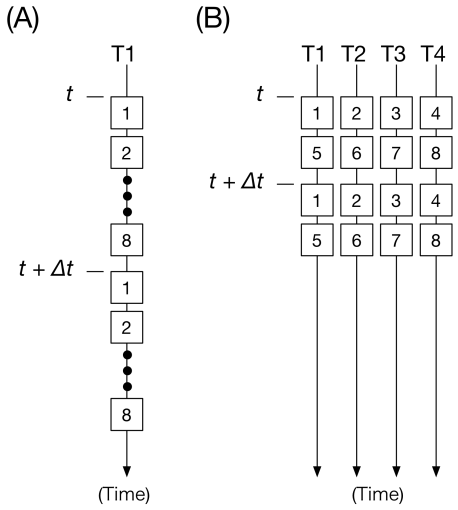

Fig. 5 Schematic of parallel computing for a neural network with 8 neurons. A: Sequential calculation by 1 thread. B: Parallel calculation by 4 threads. A square represents a nuron, and T1–4 represents threads 1–4, respectively.

Moreover, it is possible to parallelize the calculation of synaptic inputs explicitly. The following is part of the function `calculateSynapse`.

Listing 6 `ompsyn.c`

```
1  double r = 0.;
2  #pragma omp parallel for reduction(+:r)
3  for ( int j = 0; j < N_EXC; j++ ) {
4    r += w_exc [ j + N * i ] * spike [ j ];
5  }
6  ge [ i ] += DT * ( G_EXC * r - ge [ i ] ) / TAU_GE;
```

The code calculates the excitatory synaptic input $ge_i(t)$ for neuron i. We define a temporal variable `r` (line 11), and perform a dot product of synaptic connections `w_exc` $\in \{0, 1\}$ and spikes `spike` (lines 4,5) with respect to the presynaptic neurons, and store the result (lines 7,8). `#pragma omp parallel for reduction(+:r)` in line 2 describes a directive of OpenMP that performs the addition to `r` in parallel (Fig. 6). This type of calculation is called reduction.

Unfortunately, our cluster machine did not accelerate the simulation by this method [*4]

That's all for the second course. Break!

[*3] Intel CPUs have Hyperthreading function that allows 1 core to issue 2 threads, but our cluster machine disables this function. So, in this tutorial, 1 core can issues 1 thread.

[*4] A cluster machine in my lab did by the way.

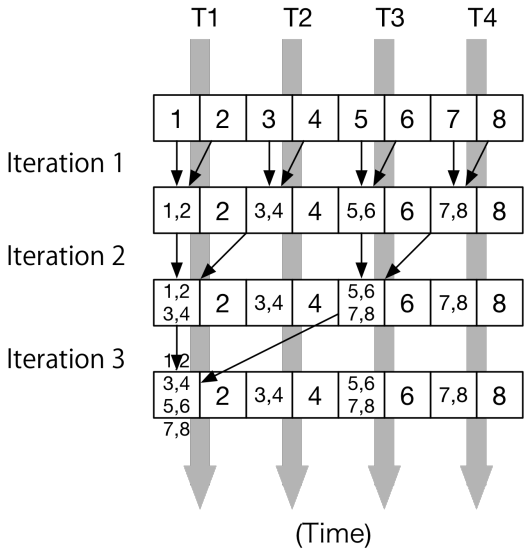

Fig. 6 Schematic of the reduction by OpenMP. When performing the addition of 8 elements using 4 threads, first each thread performs the addition of neighboring 2 elements and synchronizes. Then, 2 out of 4 threds performs the addition further and synchronizes. Finally, 1 thread performs the addition and completes the whole reduction.

6 Third course: Parallel simulation by MPI and hybrid parallelization

6.1 Using MPI

The CPU in our cluster machine for this tutorial has only 4 cores. If we want to increase the number of parallelization, we need to employ multi CPUs on multi computer nodes simultaneously. We use MPI (Message Passing Interface) for this purpose. MPI is a standard specification for parallel computing with multiple computational nodes [8], and there are several different implementations from free ones to vendor specific ones. MPI provides various directives includeing synchronous/asynchronous send/receive, synchronization or barrier, and reduction. The low-level description allows us to control the communication in fine scale, while the resulting code can become complicated. Here, we introduce the parallelization using `MPI_Allgather` directive, which would be the simplest way. The current generation of the NEST simulator [9] essentially uses this method.

The parallelized code looks like this:

Listing 7 `mpi.c`

```
void loop ( const int mpi_size, const int mpi_rank )
{
  const int n_each = N / mpi_size;
  int spike_local [ n_each ];
  timer_start ();
  for ( int nt = 0; nt < NT; nt++ ) {
    for ( int n = 0; n < n_each; n++ ) {
      calculateSynapse ( n, n_each * mpi_rank );
      updateMembranePotential ( n, n_each * mpi_rank, spike_local );
    }
    MPI_Allgather ( spike_local, n_each, MPI_INT, spike, n_each, MPI_INT, MPI_COMM_WORLD );
    if ( mpi_rank == 0 ) { outputSpike ( nt ); }
  }
```

```
14   double elapsedTime = timer_elapsed ();
15   if ( mpi_rank == 0 ) { printf ( "Elapsed time = %f sec.\n", elapsedTime); }
16 }
```

First, two constants `mpi_size`, `mpi_rank` are passed to the function (lines 1,2). These values are obtained at the initialization of MPI, and they represent the total number of threads and the thread id for itself, respectively. The loop with respect to neurons are modified largely (line 7). In the case of OpenMP, the number of iterations of this loop was N. On the other hand, in the case of MPI, the number is `n_each` $= N/$`mpi_rank`. That is, each thread calculates only `n_each` neurons. To do so, we extend the arguments of `calculateSynapse` and `updateMembranePotential`. We store the spike information to an array `spike_local` of size `n_each`, which is defined in line 4. Lines 14–16 calls `MPI_Allgather`. After the call, all the threads share the information of `spike_local`, and store the results in array `spike` of size N (Fig. 7). So, after the call, the spike information of all neurons are restored and shared by all the threads.

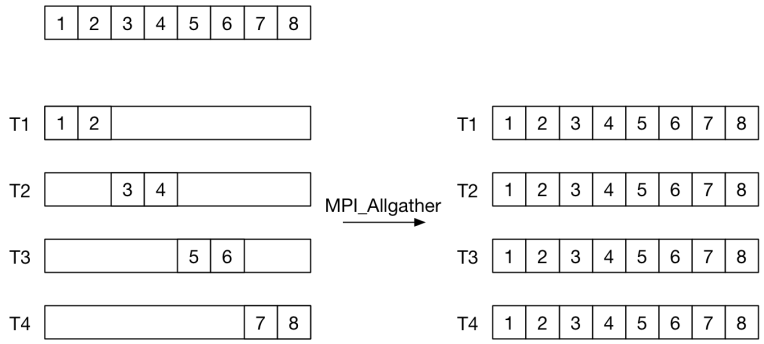

Fig. 7 Parallel calculation using `MPI_Allgather`. This example calculates 8 neurons (square) by 4 threads (T1–4). Each thread calculates only 2 neurons and holds the information of the spike generation. After the call of `MPI_Allgather` the spike information (local to each thread) is exchanged and shared by all the threads globally. That is, all threads have the necessary information to perform further calculation.

MPI codes are compiled by `mpicc`, and executed by `mpirun` as follows:

```
[tyam@plato tutorial]$ mpicc -std=c99 -O3 -Wall -o mpi mpi.c
[tyam@plato tutorial]$ mpirun -hostfile hostfile -np 16 ./mpi
```

where `hostfile` is the text file that describes the available computational nodes, and `-np` is the number of processors that we use. The content of `hostfile` looks like this:

```
1 plato01:2
2 plato02:2
3 :
4 plato16:2
```

The node name and the number of CPUs per node is concatenated by colon (:). If we use 1 cpu per node and set `-np 16`, the computational time is about 1.5 sec, indicating $25/1.5 = 16$ times speed up.

Caution! When using MPI, only one person can test at a time, because the person occupies all the computational nodes. So, keep in line[*5].

In the above code, calculation and communication are made sequentially. In very large-scale simulation, the communication cost exceeds the calculation cost. There are several ways to hide the communication cost over the calculatio cost such as [10].

[*5] Usually, we use a batch queueing system

6.2 OpenMP + MPI

We can hybrid OpenMP and MPI. Unfortunately, on our cluster machine, the hybrid organization made the calculation slower.

6.3 Flat MPI

We can use all cores in a cpu by MPI solely without OpenMP, which is called Flat MPI. We can increse the number given to `-np` option. Our cluster machine has

$$\begin{aligned} &\#\text{ nodes} \times \#\text{ cpus per node} \times \#\text{ cores per cpu} \times \#\text{ threads per core} \\ &= 16 \times 2 \times 4 \times 1 \\ &= 128 \end{aligned} \tag{12}$$

cores, so the calculation gets faster up to `-np 128`. In this case, the calculation completed by 0.3 sec.

Exercise 3.
Examine how the computation time gets shorter as the processor number (`-np`) increses (Strong scaling).

6.4 Misc

GPUs are popular hardware for parallelcomputing. CUDA (Compute Unified Device Architecture)[11] is a library that NVIDIA develops for their GPUs, and OpenCL[12] is another library specified by OpenCL Working Group.

In this tutorial, we considered only the LIF model. There are much more issues to study such as Hodgkin-Huxley model for simulation of ion channels, and cable equations for simulation of current flows within neurons composed of many compartments.

I would love to have a short course for 2-3 days with world-wide students and senior scientists for invited talks.

7 Closing

That's all for this tutorial. Well done! See you again.

References

[1] GIGAZINE (2013). Just 1 second of the human brain activity corresponds to 40 minutes of Supercomputer K (Japanese article) http://gigazine.net/news/20130806-simulating-1-second-of-real-brain/. (Last access October 14, 2017)

[2] Exascale Computing Project. https://exascaleproject.org/ (Last access October 20, 2017)

[3] Gerstner W, Kistler W. Spiking Neuron Models. Cambridge (2001).

[4] Example: CUBA. http://brian2.readthedocs.io/en/stable/examples/CUBA.html (Last access October 17, 2017)

[5] Brette R et al. Simulation of networks of spiking neurons: A review of tools and strategies. J Comp Neurosci 23:349-398, 2007.

[6] Goodman DF, Brette R. The Brian simulator. Front Neurosci 10:3389, 2009.

[7] Chandra R et al. Parallel Programming in OpenMP. Morgan Kaufmann (200).

[8] Gropp W, Lusk E, Skjellum A. Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface. MIT Press (1999).

[9] Gewaltig MO, Diesmann M (2007) NEST (Neural Simulation Tool) Scholarpedia 2(4):1430.

[10] Igarashi J (2017). Parallel computing of a spiking neural network model of a layered cortical sheet using a tile partitioning method. Annual conference of Japanese Neural Network Society (JNNS2017), Sep 20-22, 2017, Kitakyushu International Conference Center

[11] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (Last access October 27, 2017)

[12] OpenCL Working Group. OpenCL Overview. https:// www.khronos.org/opencl/ (Last access October 27, 2017)