Introduction of High-Performance Computing for Neuroinformatics

山﨑 匡 *
電気通信大学 大学院 情報理工学研究科

2017 年 11 月 21 日

概要

神経回路の数値シミュレーションは、大規模かつ精緻になるに従い、莫大な計算時間がかかるようになる。本チュートリアルでは、神経回路シミュレーションの基本事項をまずおさらいし、次いで様々な並列化によって計算を高速化する手法を学ぶ。単なる座学ではなく、実際に手を動かしてコードを書き実行させる、ハンズオンの形式を取る。

目次

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

1 はじめに

生命維持から知的活動まで、脳は様々な機能を担っているが、その計算原理は未だに明らかになっていない。一方、脳の構造はそれに比べるとよく分かっており、ニューロンと呼ばれる神経細胞が複雑に繋がりあったネットワークである。一個のニューロンの挙動は具体的に数式で記述できるので、ニューロンの個数分そのような数式をプログラムし、コンピュータで数値シミュレーションを行うことで、原理的には脳の活動をコンピュータ上に再現することが可能になる。

ヒトの脳は約 1000 億個のニューロンからなると言われている。その全てを現実的な時間でシミュレートすることは、現在の最高性能のスパコンをもってしても難しい [1, 2]。しかしより小規模な動物の脳や、脳の一部を現実的な時間でシミュレートすることは十分可能になってきている。

その際に本質的なのは、どのようにして計算を速くするか？ である。並列計算の技法を駆使することで、計算時間を数十倍から数百倍短縮することが可能になる。本チュートリアルではそのような手法を、典型的なランダムネットワークのシミュレーションを題材にして紹介する。

大体のスケジュールは以下の通り。

13:00–13:10 開会式
13:10–13:30 クラスタマシンへのログイン
13:30–14:30 一時間目: 神経回路シミュレーション事始め
14:30–14:40 休憩
14:40–15:40 二時間目: OpenMP による計算の並列化
15:40–15:50 休憩
15:50–16:50 三時間目: MPI による計算の並列化とハイブリッド並列
16:50–17:00 閉会式

このチュートリアルは、文部科学省 ポスト「京」萌芽的課題 4「思考を実現する神経回路機構の解明と人工知能への応用」の、「ボトムアップで始原的知能を理解する昆虫全脳シミュレーション」ならびに「脳のビッグデータ解析、全脳シミュレーションと脳型人工知能アーキテクチャ」[*1] の協賛でお送りしています。

2 開会式

試合開始。まあ軽く自己紹介とか？

3 クラスタマシンへのログイン

各自 ssh の公開鍵を作って私に下さい。登録して引き替えにユーザ名をお渡しします。

公開鍵の作り方は、

```
[tyam@plato tutorial]$ ssh-keygen -t rsa
```

です。パスフレーズを決めて入力すると、~/.ssh/id_rsa と~/.ssh/id_rsa.pub ができるので、id_rsa.pub を下さい。

ユーザ名をもらったら、ログインしてみてください。マシン名は plato.sim.neuroinf.jp です。

```
[tyam@plato tutorial]$ ssh plato.sim.neuroinf.jp -l <username>
```

として、<username>のところに指定されたユーザ名を記載すれば、ログインできるはず。

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

4 一時間目: 神経回路シミュレーション事始め

4.1 ニューロン 1 個のシミュレーション

まず 1 個のニューロンのシミュレーションから始めよう [3]。ニューロンの代表的なモデルは Hodgkin-Huxley モデルだが、本チュートリアルではより簡単な積分発火型モデル (Leaky integrate-and-fire model, LIF) を用いる。

カレントベースの LIF モデルは次の式で記述される。

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

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

$$v(0) = V_{\mathrm{init}}. \tag{3}$$

ここで、$v(t)$ (mV) は時刻 t での膜電位、τ (ms) は時定数、V_{leak} (mV) は静止電位、R (MΩ) は膜の抵抗、$I_{\mathrm{ext}}(t)$ (nA) は時刻 t での外部電流、θ (mV) はスパイク発射のための閾値、V_{reset} (mV) はリセット電位、V_{init} (mV) は膜電位の初期値である。

式 (1) が基本的な膜電位のダイナミクスを記述する。V_{leak} を平衡点とし、外部入力 $RI_{\mathrm{ext}}(t)$ に時定数 τ で漸近する挙動を示す。式 (2) はスパイク発射の条件である。膜電位が閾値を超えると、その時刻でスパイクを発射したものとし (Spike (t) = 1)、かつ膜電位をリセットする。式 (3) は膜電位の初期値の与える。

この微分方程式をコンピュータで数値的に解くために、差分方程式に変換する。具体的には十分短い時間間隔 Δt を考え、

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

と近似する。一方、$v(t)$ を t の回りで Δt でテイラー展開すると、

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

となり、Δt が十分小さいという仮定の下 $O(\Delta t^2)$ 以降の項を無視すると

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

となる。最後に上記 2 式を組み合わせると、

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

となる。ここで

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

である。後は初期値 $v(0)$ さえ与えられれば、式 (7) を $t = 0, \Delta t, 2\Delta t, \cdots$ と逐次的に計算することで、$v(t)$ の値を近似的に求めることができる。この数値解法には陽的オイラー法という名前がついている。

これを実際に試してみよう。次のコードを試す。

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  }
```

このコードでは、1000 ミリ秒 =1 秒) 間のシミュレーションを $\Delta t = 1$ ミリ秒で行う。100 ミリ秒から 900 ミリ秒までの間、外部電流として $I_{\rm ext} = 12$ nA を与える。膜抵抗は簡単のために 1 MΩ とする。このときの $v(t)$ を、初期値 $v(0) = -65$ mV から Δt 毎に逐次的に計算する。

コードの実行は 29 行目の `main` から始まり、関数 `loop` を実行するだけである (31 行目)。よって関数 `loop`(14–27 行目) がシミュレーションのコードそのものである。関数 `loop` の中身を詳しく見ていく。

16 行目で膜電位の変数 `v` を定義し、初期値として `V_INIT`= −65 mV を代入する。`V_INIT` の定義は 5 行目である。

17 行目が時間に関するループである。シミュレートする時間を $T = 1000$ ミリ秒間とし (10 行目)、それを $\Delta t =$ `DT` $= 1$ ミリ秒の刻み (9 行目) で計算するので、ループの回数 `NT` は `NT` $= T/DT = 1000$ 回である。変数 `nt` を用意してカウントする。

18 行目で、今の時刻での $v(t)$ の値を、時刻と共に表示する。

19 行目で、外部電流の値を設定する。3 項演算子を使って、100 ミリ秒から 900 ミリ秒までの間、`i_ext`=`I_EXT`、それ以外は 0 とする。`I_EXT` は 12 行目で定義されている。

20 行目で、式 (8) に従って $\Delta v(t)$ を計算する。τ は `TAU` $= 20$ms として 3 行目で、$R_M =$ `R_M` $= 1$ MΩ は 8 行目でそれぞれ定義されている。

21 行目で、式 (7) に従って $v(t)$ を更新する。

22–25 行目はスパイク発射の判定である。もし $v(t)$ が閾値 `THETA` を越えていたら (22 行目)、膜電位として 0 mV をその時刻と共に表示し (23 行目)、$v(t)$ を `V_RESET` にセットする。`THETA` $= -55$ mV は 7 行目で、`V_RESET` $= -65$ mV は 7 行目でそれぞれ定義されている。

このコードをコンパイルして実行してみよう。コンパイルは以下のようにする。

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

`-Wall` オプションは、全ての警告を表示するもので、超推奨。正常にコンパイルできると実行ファイル `lif` ができるので、以下のように実行する。

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

実行すると、数字がどばっと表示されたと思うが、それが各時刻とその時の $v(t)$ の値である。数字を眺めても何もわからないので、以下のようにリダイレクトしてファイルに出力し、

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

gnuplot で表示する。

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

すると、図 1 のような膜電位の表示が得られるはずである。100–900 ミリ秒の間、一定の間隔でスパイクを発射している様子が確認できた。

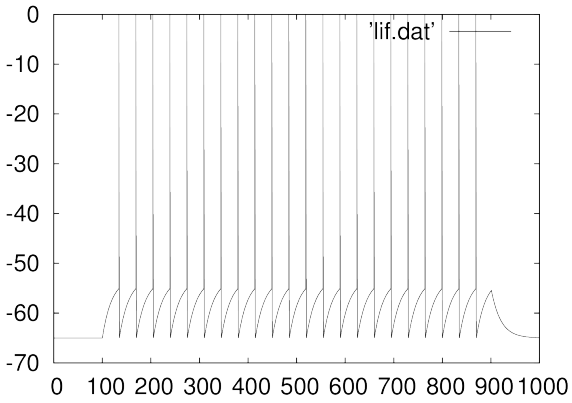

図 1　1 個のニューロンの膜電位のプロット

ここで注意！　パラメータの単位には気をつけること。例えばもしこのプログラムで時間をミリ秒ではなく秒にして、$T = 1.0$, $\Delta t = 0.001$ とすると、正しい計算が行われない。このプログラムのように Physiologial Unit を使うか、あるいは SI Unit を使うか、どちらかにすること。

4.2　ニューロン 2 個のシミュレーション

一番簡単なネットワークはニューロン 2 個からなるものなので、次はそれを作ろう。

ニューロン同士がシナプスで繋がっておらず、完全に独立な場合は、`lif.c` をベースにしてほとんど自明に書ける。具体的には変数 `v`, `dv` を配列にして変数 `v[2]`, `dv[2]` とすれば良い。ただしそれだけでは完全に同じ計算をするだけなので、`v` の初期値を片方は 10 mV 下げよう。コードは以下のようになる。

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  }
```

コードの変更点は以下の通りである。v を配列にし (16 行目)、初期値を変更した。外部電流は 0 ミリ秒から入れることとした (17 行目)。dv も配列にした (20 行目)。dv, v の計算は添字を変えて 2 回計算した (21–24 行目)。閾値を超えたときの表示の仕方を変えた (26,30 行目)。

本来であれば 2 個のニューロンの計算は for ループで回すべきであるが、自明さを示すためにわざとループで書かなかった。

これをコンパイルして実行し、結果をプロットする。

```
[tyam@plato tutorial]$ gcc -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
```

図 2 のような膜電位のプロットが得られるはずである。初期状態が異なるのでスパイクのタイミングはずれるが、その他は同じなので同じ波形がシフトするだけとなる。

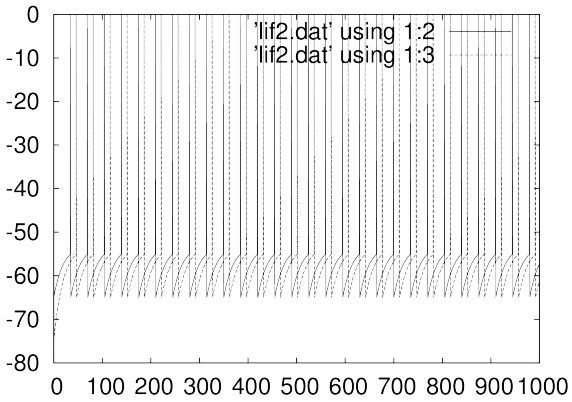

図 2　2 個の独立なニューロンの膜電位のプロット

4.3　ネットワークのシミュレーション

一時間目の最後に、2 個のニューロンをシナプスで結合してちゃんとしたネットワークにしよう。ここでは一番簡単な exponential synapse を導入する。

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

膜電位の式の右辺にシナプス後電位 $g(t)$ を追加する。ここで、i はニューロンの番号 ($i \in \{0, 1\}$)、τ_{syn} は時定数、$g_i(t)$ はシナプス後電位、w は結合重み、$\mathrm{Spike}_{(i+1)\%2}(t)$ はもう片方のニューロンのスパイク発射 (0 または 1) を表す[*2]。これも同様に差分化し、陽的オイラー法で解く。以下のようにすればよい。

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

ただし、初期値 $g_i(0)$ は 0 mV とする。この $g_i(t)$ を、膜電位の式の右辺に追加する。

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. };
19   double i_ext = I_EXT;
```

[*2] $i = 0$ のとき $(i+1)\%2 = 1$、$i = 1$ のとき $(i+1)\%2 = 0$ なので。

```
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 }
```

コードの変更点は以下の通りである。まずシナプス入力の変数 g(20 行目) とスパイクの変数 spike(21 行目) を定義する。25,26 行目で膜電位の式にシナプス入力を加える。28,29 行目でシナプス入力の計算を行い、32,33 行目で値を更新する。34,39 行目でスパイクを発射したかどうかの判定を行う。

これをコンパイルして実行し、結果をプロットする。

```
[tyam@plato tutorial]$ gcc -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
```

図 3 のような膜電位のプロットが得られるはずである。今度はスパイクのタイミングが徐々に揃って行くことがわかる。

課題 1.

互いを抑制性で繋ぐと何が起こるか試して確認せよ。具体的には W の値の符号を負にすればよい。

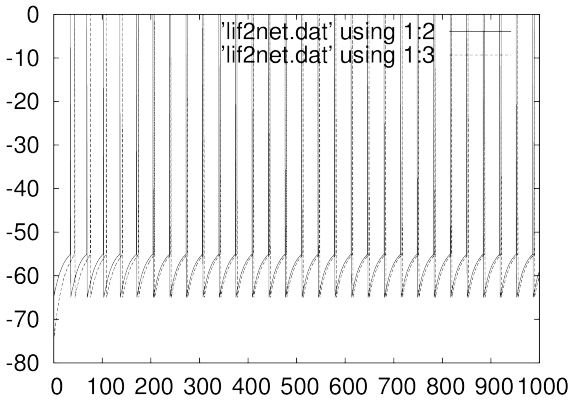

図 3　互いに興奮性で接続された 2 個のニューロンの膜電位

課題 2.

なぜこうなるのかを考察せよ。

一時間目はここまで。休憩！

5 二時間目: OpenMP による計算の並列化

5.1 ランダムネットワークのシミュレーション

一時間目にやった 2 個のニューロンからなるネットワークは小さすぎて、計算があっという間に終わってしまった。これでは面白くないので、もう少し大きなネットワークを考えよう。具体的には 4000 個のニューロンを 4:1 で興奮:抑制に振り分け、確率 $p = 0.02$ でランダムに結合させた、ランダムネットワークを考える [4]。このネットワークは様々な神経回路シミュレータのベンチマークとしても利用されている、スタンダードなものである [5]。

膜電位の式は式 (1)–(3) と同様である:

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

ここで、$v(t)$ は時刻 t での膜電位、$\tau = 20$ ms は時定数、$V_{\text{leak}} = -49$ mV は静止電位、$ge(t)$, $gi(t)$ はそれぞれ興奮性、抑制性のシナプス電流、$\theta = -50$ mV はスパイク発射のための閾値、$V_{\text{reset}} = -60$ mV はリセット電位、$V_{\text{init}} = -60 + 10 \times \text{rand}(t)$ は膜電位の初期値、$\text{rand}(t)$ は $[0, 1)$ の一様乱数である。一方、シナプス電流は以下の式で計算する。

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

ここで、$\tau_e, \tau_i = 5, 10$ ms はそれぞれ時定数、Exc, Inh はそれぞれ興奮性、抑制性のニューロン集団、$w_e, w_i = +1.62, -9$ mV はそれぞれスパイク入力 1 発あたりのシナプス後電位の変化量、$\text{Spike}_j(t) \in \{0, 1\}$ はニューロン j が時刻 t でスパイクを発射した場合 1, そうでなければ 0 である。

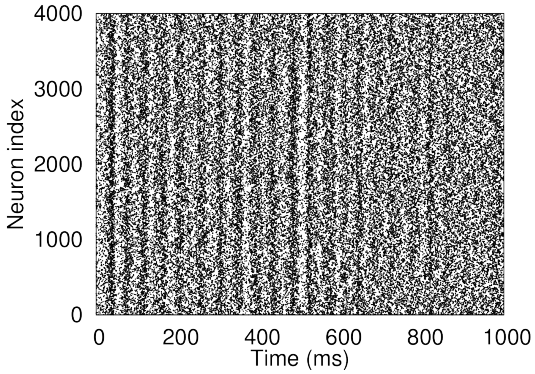

図 4 ランダムネットワークのラスタープロット。横軸は時間 (ms)、縦軸はニューロン番号である。1 つのドットが 1 つのスパイクを表す。

コードは付録にある。コンパイルして実行すると、`spike.dat` というファイルが生成されて、`gnuplot` で表示すると図 4 のようなラスタープロットが得られる。4000 個のニューロンの膜電位を一度にプロットしてもまともに見えないので、以降はこのようにスパイクだけをプロットする。

コードの概要は以下の通りである。

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 }
```

時間に関するループ (3 行目) はこれまで通り。ニューロンの数が増えたので、今回からちゃんとループにする (4 行目)。ループの中では、まずシナプス入力の計算をし (5 行目)、ついで膜電位の値を更新する (6 行目)。ニューロンの計算が終わったら、スパイクの情報をファイルに出力する (8 行目)。関数 `calculateSynapse` および `updateMembranePotential` の中身はご想像の通りである。

本チュートリアルで使うクラスタマシンを普通に使って計算すると、1 回のシミュレーションに 25 秒かかる。各自試してみよ。

5.2 ニューロンの計算の並列化

ニューロン数やシナプス数が多くなるにつれ、計算時間は膨大になる。これを解決するための手法は、CPU のマルチコア機能や複数 CPU を同時に利用するためのライブラリ、あるいはグラフィクスプロセッシングユニット (GPU) に代表されるアクセラレータとよばれるハードウェアを使った並列計算である。一般的には、ニューロンのループ (4 行目とその内側) を分割して、ニューロン毎に独立に並列計算する。

5.3 OpenMP

OpenMP は 1 個 CPU に含まれる複数の計算コアを並列に使うための標準規格であり [7]、GNU Compiler Collection (GCC) の C コンパイラには標準的についている機能である。ソースコード中のヘッダ領域で`#include <omp.h>`し、コンパイル時に`-fopenmp` オプションを追加すると、OpenMP を利用する準備が整う。

ニューロンのループの並列化は容易である。以下のようにループの直前に 1 行追加するだけである。

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 }
```

4 行目の`#pragma omp parallel for` が OpenMP の命令である。これを記述すると、自動的にその次の行の `for` ループ、つまり各ニューロンの計算がニューロン毎に並列実行される。ニューロン毎の計算は独立なので、この部分は完全に並列実行可能である。

並列化のイメージは図 5 の通りである。1 CPU・1 コアの場合は、1 つのスレッド (計算単位) がニューロンの計算を逐次的に実行する (図 5A)。OpenMP を使うと CPU の複数コアを同時に利用し、複数のスレッドでニューロンの計算を個別に実行する (図 5B)。よって、理論上は計算時間はスレッド数倍高速になる。

この計算を 4 コア全部[*3]使って行うと、計算時間は 12.5 秒に短縮され、25/12.5 = 2 倍高速化される。

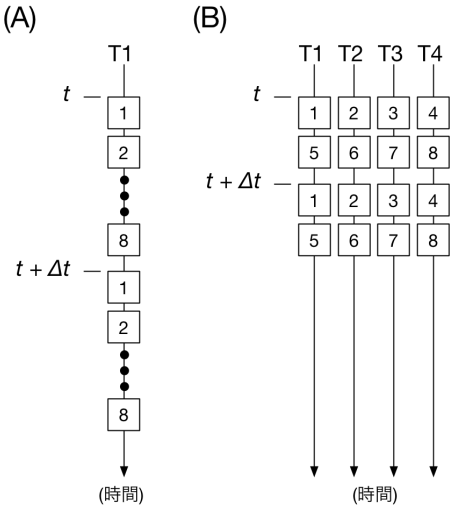

図 5　ニューロン 8 個からなるネットワークの並列計算イメージ。A: 1 スレッドによる逐次実行の場合。 B: 4 スレッドによる並列実行の場合。ニューロンを四角で、スレッド 1–4 を T1–4 でそれぞれ表す。

また、シナプス入力の計算部分を明示的に並列化することも可能である。`calculateSynapse` 関数の内容を一部抜粋して掲載する。

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;
```

ニューロン i の興奮性シナプス入力 $ge_i(t)$ を計算する。中間変数 `r` を用意して (1 行目)、プレ側のニューロンについて、シナプス結合の有無 `w_exc` $\in \{0, 1\}$ とスパイク `spike` の積和を計算し (4,5 行目)、結果を格納する (7,8 行目)。2 行目の`#pragma omp parallel for reduction(+:r)` が OpenMP の命令で、変数 `r` への加算を並列に実行する (図 6)。このような計算をリダクションと言う。

本チュートリアルで使うクラスタマシンでは、残念ながらこれを加えても計算時間が変わらなかった[*4]。

二時間目はここまで。休憩！

[*3] Intel の CPU には Hyperthreading という機能があり、1 コアで 2 スレッド発行可能であるが、本チュートリアルで使うクラスタマシンではこの機能は disable である。よって 1 コア =1 スレッドとなっている。

[*4] 私の研究室のクラスタでは速くなった。

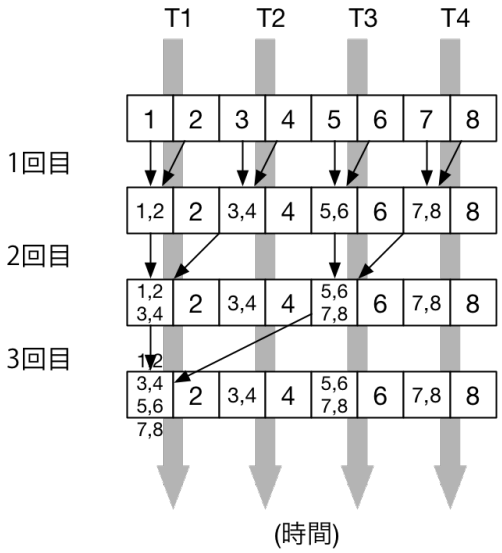

図 6　OpenMP によるリダクションの概略。4 スレッドで 8 個の要素の加算を実行する場合、まず各スレッドは隣接する 2 要素の加算を実行して同期を取る。次に 2 スレッドだけが計算結果をさらに加算して同期する。最後に 1 スレッドだけがさらに加算し、全体のリダクションが完成する。

6　三時間目: MPI による計算の並列化

本チュートリアルで使うクラスタマシンでは 1 CPU あたり 4 コアしか搭載していないので、これ以上並列度をあげたければ複数の CPU・計算ノードを同時に使う必要がある。それは MPI (Message Passing Interface) を用いて行う。複数の計算ノードを用いた並列計算のための標準規格であり [8]、フリーなものからネットワーク機器ベンダの独自のものまで、様々な実装がある。MPI には同期/非同期の送信/受信、全体の制御の同期、リダクションなど様々な命令が用意されていて、非常に低レベルの記述ができる分、コードは複雑になる傾向がある。ここでは最も容易な `MPI_Allgather` 命令を用いた並列化を紹介する。例えば現世代の NEST シミュレータ [9] も本質的に同じ並列化を行っている。

並列化されたコードを次に示す。

Listing 7　`mpi.c`

```
1  void loop ( const int mpi_size, const int mpi_rank )
2  {
3    const int n_each = N / mpi_size;
4    int spike_local [ n_each ];
5    timer_start ();
6    for ( int nt = 0; nt < NT; nt++ ) {
7      for ( int n = 0; n < n_each; n++ ) {
8        calculateSynapse ( n, n_each * mpi_rank );
9        updateMembranePotential ( n, n_each * mpi_rank, spike_local );
10     }
11     MPI_Allgather ( spike_local, n_each, MPI_INT, spike, n_each, MPI_INT, MPI_COMM_WORLD );
12     if ( mpi_rank == 0 ) { outputSpike ( nt ); }
13   }
14   double elapsedTime = timer_elapsed ();
15   if ( mpi_rank == 0 ) { printf ( "Elapsed␣time␣=␣%f␣sec.\n", elapsedTime); }
16 }
```

まず、関数に引数 mpi_size, mpi_rank が渡されている (1,2 行目)。これは MPI を初期化したときに得られる値であり、それぞれスレッドの総数と、自分自身のスレッド番号を表す。大きな変更点はニューロンに関するループ (7 行目) である。OpenMP による並列化の場合は、ループの回数は全ニューロン数である N であった。今回は全ニューロン数を全スレッド数で割り、各スレッドは n_each $= N/$mpi_rank 回ループして、スレッド毎に n_each 個のニューロンのみを計算する。そのため、関数 calculateSynapse と updateMembranePotential の引数に、計算すべきニューロン群の先頭の番号 (n_each * mpi_rank) を加える。計算されたスパイク発射の情報は大きさ n_each の配列 spike_local に格納する。spike_local の定義は 4 行目である。MPI_Allgather の実行は 14–16 行目である。これを実行すると、各スレッドが保持している spike_local の内容を全スレッドで共有し、結果を大きさ N の配列 spike に格納する (図 7)。よって命令の実行後は逐次版あるいは OpenMP 版と同様に、ニューロンのスパイクの情報が復元され、全スレッドで共有されることになる。

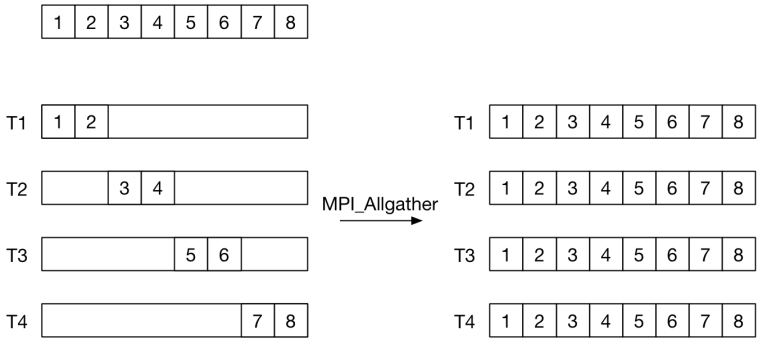

図 7　MPI_Allgather による計算の並列化。4 スレッド (T1–4) で 8 ニューロン (四角) の計算をする例を考える。各スレッドは 2 個のニューロンの計算だけを行い、スパイク発射の有無を保持する。MPI_Allgather を実行するとスパイクの情報が交換され、全スレッドで共有される。実行後は全てのスレッドが逐次計算版と同じ状態になり、計算を継続できる。

MPI を使ったコードは mpicc でコンパイルし、mpirun で以下のように実行する[*5]。

```
[tyam@plato tutorial]$ mpicc -O3 -Wall -o main main.c
[tyam@plato tutorial]$ mpirun -hostfile hostfile -np 32 ./main
```

hostfile は利用可能な計算ノード名を記載したテキストファイル、-np の引数は実際に計算に使う CPU 数である。hostfile の内容は例えば次のようになる。

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

計算ノード名とノード当たりの CPU 数をコロン (:) で繋いだものを列挙する。1 ノード 1CPU ずつ使い -np 16 とすると、計算時間は 1.5 秒にまで短縮された。$25/1.5 = 16$ 倍の高速化である。

ここで注意！　MPI による並列シミュレーションでは一人で全ノードを占有することになるので、一度に一人しか試すことができない。声をかけあって順番を守ること[*6]。

なお、上記コードでは計算と通信を順番に行っているが、大規模計算で多数の計算ノードが関わる場合は、計算よりも通信に時間がかかるようになる。その解消法の話題は高度なので別の機会にゆずるが、例えば [10] などがある。

[*5] MPI が MVAPICH2 の場合

[*6] そういうわけで普通はキューイングシステムを使う。今回はできるだけ生に近い状態で試したいので、そういうものは使わない。

6.1 OpenMP + MPI

もちろん OpenMP と MPI は両方同時に利用してハイブリッドにできる。残念ながら本チュートリアルで使うクラスタマシンでは、ハイブリッドにするとかえって遅くなってしまった。興味があれば試してみよ。

6.2 フラット MPI

では MPI では CPU 1 コアあたり 1 スレッドしか使えないのかというとそんなことはなくて、単に`-np`の引数の値を大きくすれば良い。本チュートリアルで使うクラスタマシンでは、

$$\begin{aligned} &\text{ノード数} \times 1 \text{ ノードあたりの CPU 数} \times 1\text{CPU あたりのコア数} \times 1 \text{ コアあたりのスレッド数} \\ &= 16 \times 2 \times 4 \times 1 \\ &= 128 \end{aligned} \tag{12}$$

なので、`-np 128` までは速くなる。実際`-np 128` としたときの実行時間は 0.3 秒となった。

課題 3.

`-np` の値をいくつか試して計算時間がどう変わるかを調べよ。理想的には例えば`-np` の値を 2 倍にすると計算時間は 1/2 になり、一般に n 倍すると $1/n$ になる。このような理想的な状態を強スケーリングと言う。

6.3 その他

並列計算用のハードウェアとして GPU が良く用いられる。NVIDIA 社が自社の GPU 用に開発している並列計算ライブラリ CUDA (Compute Unified Device Architecture)[11]、OpenCL Working Group が仕様策定しているライブラリ OpenCL[12] で並列化されたコードが書けるが、本チュートリアルの範囲を超えるので、今回はやらない。

また今回はニューロンモデルとして LIF だけを考えたが、スパイク生成のメカニズムを研究するなら Hodgkin-Huxley 方程式を解く必要があるし、ニューロンの形状を研究するならマルチコンパートメントモデルにする必要があり、その場合はケーブル方程式を解く必要がある。それらについても本チュートリアルの範囲を超える。

3 日間くらいのショートコースを開催するのはどうでしょうね。海外からも学生を呼んで、偉い人の招待講演もつけて。

7 閉会式

試合終了。またどこかでお会いしましょう。

参考文献

[1] GIGAZINE (2013). 人間の脳の活動でわずか 1 秒間はなんとスーパーコンピュータ「京」の 40 分に匹敵することが判明 http://gigazine.net/news/20130806-simulating-1-second-of-real-brain/. (最終アクセス 2017 年 10 月 14 日)

[2] Exascale Computing Project. https://exascaleproject.org/ (最終アクセス 2017 年 10 月 20 日)

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

[4] Example: CUBA. http://brian2.readthedocs.io/en/stable/examples/CUBA.html (最終アクセス 2017 年 10 月 17 日)

[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). Paralel computing of a spiking neural network model of a layered cortical sheet using a tile partitioning method. 日本神経回路学会全国大会, 北九州国際会議場, 小倉.

[11] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (最終アクセス 2017 年 10 月 27 日)

[12] OpenCL Working Group. OpenCL Overview. https:// www.khronos.org/opencl/ (最終アクセス 2017 年 10 月 27 日)