Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Параллельные численные методы»

Лабораторная работа
Численное решение систем обыкновенных дифференциальных уравнений на примере решения задачи моделирования мозга

Мееров И.Б., Сысоев А.В., Комаров М.А.

При поддержке компании Intel

Нижний Новгород
2011

Содержание

Введение

Обыкновенные дифференциальные уравнения (ОДУ) и системы ОДУ находят активное применение при построении математических моделей различных явлений и процессов. Несмотря на то, что для решения ОДУ и систем ОДУ, удовлетворяющих некоторым требованиям, разработаны мощные аналитические методы, на практике весьма часто приходится прибегать к численному решению. Расчеты нередко оказываются трудоемкими, что обуславливает актуальность постановки вопроса об использовании параллельных вычислений. К числу факторов, затрудняющих разработку эффективных параллельных алгоритмов, можно отнести итерационную природу методов решения ОДУ и систем ОДУ, наличие зависимостей между уравнениями в системах, недостаточную трудоемкость вычисления правых частей и др. В общем случае гарантировать близкую к линейной масштабируемость соответствующего параллельного алгоритма не представляется возможным. Попыткам решения проблемы посвящено немало научной литературы. Желающие могут ознакомиться, например, с работами [8-10]. В данной лабораторной работе дается краткое введение в методы численного интегрирования систем ОДУ на примере задачи моделирования мозга, описывается схема распараллеливания, обсуждаются вопросы производительности, приводятся результаты вычислительных экспериментов.

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – введение в параллельные методы численного решения систем обыкновенных дифференциальных уравнений.

Данная цель предполагает решение следующих основных задач:

1. Краткое введение в проблематику моделирования мозга.
2. Рассмотрение метода Эйлера для численного решения систем ОДУ.
3. Выполнение последовательной и параллельной программной реализации метода Эйлера.
4. Выполнение вычислительных экспериментов с целью практического изучения вопроса об эффективности параллельной реализации.

1.2. Структура работы

Работа построена следующим образом: дано краткое введение в проблематику моделирования мозга, сформулирована математическая модель в виде системы ОДУ, кратко описан метод Эйлера для решения систем ОДУ, выполнена его программная реализация для случая последовательного исполнения. Далее описана параллельная реализация с использованием OpenMP, проведен анализ масштабируемости, выявлены пути для ее повышения, внесены изменения в программный код, приведены результаты и выводы.

В ходе рассмотрения материала затрагиваются вопросы генерации псевдослучайных чисел с использованием библиотеки Intel MKL и возможность их использования для параллельных вычислений. Подробнее данная тема изучается в работах по численному интегрированию стохастических дифференциальных уравнений и главе «Методы Монте-Карло».

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Процессор	2 четырехъядерных процессора Intel Xeon E5520 (2.27 GHz)
Память	16 Gb
Операционная система	Microsoft Windows 7
Среда разработки	Microsoft Visual Studio 2008
Компилятор, профилировщик, отладчик	Intel Parallel Studio XE
Математическая библиотека	Intel MKL v. 10.2.5.035

1.4. Рекомендации по проведению занятий

Для выполнения лабораторной работы рекомендуется следующая последовательность действий.

1. Дать вводную информацию об ОДУ, системах ОДУ и методах их численного решения.
2. Поставить задачу моделирования мозга, сформулировать математическую модель и метод Эйлера для численного решения задачи.

3. Выполнить последовательную и параллельную программную реализацию.
4. Провести эксперименты, проанализировать их результаты. Выявить причины некоторого падения ускорения при увеличении числа ядер, сформулировать пути повышения эффективности. Выполнить программную реализацию. Вновь провести эксперименты, проанализировать их результаты.
5. Сформулировать выводы, дать задания для самостоятельной работы слушателей.

2. Введение и описание модели

Методы математического моделирования и нелинейной динамики являются мощным инструментарием для решения широкого круга задач из различных областей науки и техники. Традиционно данный инструментарий применяется в различныхзадачах теоретической и прикладной физики. Однако за последние двадцать лет существенно возрос интерес и потребность в применении математического моделирования и теории нелинейных колебаний в анализе сложных биологических систем. Одной из биологических областей, где успешно применяется нелинейнодинамический подход, является нейродинамика и науки о мозге[1]. Интерес физиков и математиков в данной области связан, прежде всего, с большим объемом накопленных экспериментальных электрофизиологических данных и отсутствием целостной теории функционирования нервной системы даже самых простейших животных. Между тем, понимание принципов работы мозга и обработки информации нервной системой может способствовать осуществлению качественного скачка в технологиях создания искусственных интеллектуальных устройств, в разработке мозг-машинных интерфейсов и многом другом.

Человеческий мозг состоит из порядка 10^{11} нейронов – электрически активных клеток, которые в результате взаимодействия друг с другом обеспечивают такие сложные когнитивные функции как память, обучение, моторный контроль и т.д [1, 2]. Нейроны «общаются» друг с другом посредством синаптических связей, по которым протекают нервные импульсы – электрические разряды, которые представляют собой достаточно сильные и резкие изменения разности потенциалов внутри и вне клетки (т.н. трансмембранного потенциала V). На сегодняшний день в теоретической и вычислительной нейронауке наиболее используемым классом моделей нейронной активности являются модели в форме обыкновенных дифференциальных уравнений, которые по своей сути описывают протекание ионных токов через клеточную мембрану нейрона и изменение трансмембранного потенциала. Данный класс называется моделями типа Ходжкина-

Хаксли. Все модели типа Ходжкина-Хаксли строятся по одинаковому принципу, в основе динамики лежит уравнение, описывающее изменение трансмембранного потенциала [3]:

$$C_m \frac{dV_i}{dt} = I_l + I_{Na} + I_K + I_i^{ext} + I_i^{sin}, \qquad i = 1, \dots, N. \tag{1}$$

Здесь V_i – потенциал нейрона с номером i, C_m – параметр, определяющий емкость мембраны, I_l – ток утечки, I_{Na} – ток, обусловленный протеканием ионов натрия через мембрану, I_K – ток, обусловленный протеканием ионов калия через мембрану, I_i^{ext} – параметр, определяющий внешний ток, I_i^{sin} – синаптический ток, возникающий в результате взаимодействия между нейронами, N – общее число нейронов в сети. Ионные токи каждого отдельного нейрона записываются следующим образом:

$$I_{Na} = m_i^3 h_i g_{Na}(V_{Na} - V_i), \;\; I_K = n_i^4 g_K(V_K - V_i), \;\; I_l = g_l(V_l - V_i).$$

Здесь $V_{Na}, V_K, V_l, g_{Na}, g_K, g_l,$ являются параметрами модели (константы), которые определяют равновесные потенциалы ионнных каналов, а также их максимальные проводимости соответственно. Сложная динамика в модели возникает в результате того, что проводимости ионных каналов имеют нелинейную зависимость от трансмембранного потенциала. Для описания данного эффекта в модель введены так называемые воротные переменные $m_i, h_i, n_i,$ изменение во времени которых описываются дифференциальными уравнениями первого порядка:

$$\begin{aligned} \frac{dm_i}{dt} &= \alpha_m(V_i)(1 - m_i) - \beta_m(V_i)m_i, \\ \frac{dh_i}{dt} &= \alpha_h(V_i)(1 - h_i) - \beta_h(V_i)h_i, \\ \frac{dn_i}{dt} &= \alpha_n(V_i)(1 - n_i) - \beta_n(V_i)n_i. \end{aligned} \tag{2}$$

Функции $\alpha_m(V_i), \alpha_h(V_i), \alpha_n(V_i), \beta_m(V_i), \beta_h(V_i), \beta_n(V_i)$ являются нелинейными функциями, зависящими от потенциала V_i. Таким образом, в случае $I_i^{sin} = 0$ уравнения (1), (2) описывают динамику отдельного нейрона. Рассмотрим теперь взаимодействие между нейронами. Как уже говорилось, нейроны связываются друг с другом посредством синаптических связей и передающихся по ним нервных импульсов. Формально в модели взаимодействие описывается с помощью слагаемого I_i^{sin} в правой части уравнения (1):

$$I_i^{sin} = \sum_j G_{ij} r_j(t)\big(V_{syn} - V_i\big), \qquad j \neq i, \qquad j = 1, \dots, N$$

Константа G_{ij} определяет максимальную проводимость синаптической связи от нейрона с номером j к нейрону с номером i, V_{syn} является параметром в модели, определяющим равновесный потенциал синаптической связи. Динамические переменные r_i описывают активацию синаптической связи в результате изменения мембранного потенциала V_i:

$$\frac{dr_i}{dt} = \alpha F(V_i)(1 - r_i) - \beta r_i, \quad (3)$$

где $F(V_i) = \frac{1}{1+e^{\frac{10-V_i}{2}}}$ – нелинейная активационная функция; α, β – константы.

Таким образом, система из $5N$ обыкновенных дифференциальных уравнений (1), (2), (3) описывает динамику нейронной сети, состоящей из N элементов, взаимодействующих друг с другом посредством синаптических связей.

3. Метод Эйлера для решения систем ОДУ

Пусть требуется найти решение задачи Коши для системы обыкновенных дифференциальных уравнений первого порядка, записанной в векторной форме следующим образом:

$$U' = F(X, U), U(x^0) = U^0. \quad (4)$$

К векторному дифференциальному уравнению (4), в принципе, можно применить любой из численных методов, изучавшихся в разделе «Методы решения систем обыкновенных дифференциальных уравнений». В данной работе применяется метод Эйлера:

$$V^{j+1} = V^j + hF\left(x^j, V^j\right), \quad (5)$$

где верхним индексом обозначен номер итерации метода, вектор V^j является приближенным значением точного решения $U\left(x^j\right)$, h – шаг метода. Подробнее метод Эйлера и вопросы его применения рассматриваются в лекциях в соответствующем разделе.

4. Программная реализация и результаты экспериментов

4.1. Последовательная версия

Выполним программную реализацию метода Эйлера для решения системы уравнений (1), (2), (3).

Прежде всего, создадим новое **Решение** (**Solution**), в которое включим первый **Проект** (**Project**) данной лабораторной работы. Последовательно выполните следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2008**.
- В меню **File** выполните команду **New→Project…**.
- В диалоговом окне **New Project** в типах проекта выберите **Win32**, в шаблонах **Win32 Console Application**, в поле **Solution** введите **ODE**, в поле **Name** – **01_Serial**, в поле **Location** укажите путь к папке с лабораторными работами курса – **c:\ParallelCalculus**. Нажмите **OK**.
- В диалоговом окне **Win32 Application Wizard** нажмите **Next** (или выберите **Application Settings** в дереве слева) и установите флаг **Empty Project**. Нажмите **Finish**.
- В окне **Solution Explorer** в папке **Source Files** выполните команду контекстного меню **Add→New Item…**. В дереве категорий слева выберите **Code**, в шаблонах справа – **C++ File (.cpp)**, в поле **Name** введите имя файла **main_s**. Нажмите **Add**.

В результате выполненной последовательности действий в окне редактора кода **Visual Studio** будет открыт пустой файл **main_s.cpp**.

Также выполним переход к использованию компилятора **Intel C++**. Для этого в окне **Solution Explorer** выберите проект **01_Serial** и выполните команду контекстного меню **Intel Parallel Composer→Use Intel C++…**. В диалоговом окне **Confirmation** нажмите **OK**.

Задачи, которые нам необходимо решить в функции **`main()`**, – инициализировать исходные и вспомогательные данные, выполнить решение методом Эйлера (с измерением времени) и вывести результаты в файл, чтобы в дальнейшем можно было провести их содержательную интерпретацию. Таким образом, функция **`main()`** будет выглядеть так:

```
int main(int argc, char* argv[])
{
  clock_t t1, t2;
```

```
  Init();

  t1 = clock();
  Solve(20000);
  t2 = clock();

  Print();

  printf("time = %f\n", double(t2 - t1) / CLOCKS_PER_SEC);

  return 0;
}
```

Параметр, который мы передали в функцию **`Solve()`**, задает количество итераций по времени.

При численном решении системы (1)–(3), состоящей из 5N уравнений, методом Эйлера необходимо работать с двумя векторами размерности 5N, соответствующими значению искомой функции в текущий и предыдущий моменты времени. В случае если уравнения системы независимы друг от друга, мы можем вдвое сократить затраты памяти в программе, записывая только что вычисленное текущее значение на место предыдущего (*x*[*i*] = *x*[*i*] + *DeltaXi;*). В данном случае не все так просто. Для реализации численного решения создадим 8 массивов, размерность которых определяется числом нейронов *N*.

```
const int N = 1000;
const double dt = 0.01;

double *G = new double[N * N];

double Vold[N], Vnew[N];
double m[N], h[N], n[N];
double I_ext[N];
double rold[N], rnew[N];
```

Отметим, что по две копии для массивов *V* и *r* необходимы в силу того, что на каждой итерации значения их элементов используются и как исходные данные (в правых частях формул (1) и (3)) и как результаты операций (в левых частях). Таким образом, в процессе выполнения каждой итерации мы будем сохранять новые значения в массивы **`Vnew`** и **`rnew`**, а в конце итерации копировать их в массивы **`Vold`** и **`rold`** соответственно. Также отметим, что константа **`dt`** определяет шаг по времени.

Инициализацию данных выполним следующим образом.

```
const int seed = 1;

void Init()
{
  int i, j;
```

```
  for (i = 0; i < N; i++)
  {
    rold[i] = m[i] = n[i] = h[i] = 0.0;
    for (j = 0; j < N; j++)
      G[i * N + j] = 0.001;
    Vold[i] = -9.0;
  }
  srand(seed);
}
```

Функция **`srand()`** необходима, чтобы инициализировать генератор псевдослучайных чисел, при помощи которого мы будем имитировать случайный шум в параметре I^{ext}.

В соответствие с (5) выполним реализацию метода Эйлера в функции **`Solve()`**.

```
void Solve(int Time)
{
  int i, count;

  // Цикл по времени
  for (count = 0; count < Time; count++)
  {
    // Цикл по нейронам
    for (i = 0; i < N; i++)
    {
      I_ext[i] = -10.0 - (double)rand() /
                 ((double)RAND_MAX);
      Vnew[i] = Vold[i] + F_v(i, &G[i * N], Vold[i], rold,
                m[i], h[i], n[i], I_ext[i]) * dt;
      m[i] = m[i] + F_m(Vold[i], m[i]) * dt;
      h[i] = h[i] + F_h(Vold[i], h[i]) * dt;
      n[i] = n[i] + F_n(Vold[i], n[i]) * dt;
      rnew[i] = rold[i] + F_r(Vold[i], rold[i]) * dt;
    }
    // Копирование
    for (i = 0; i < N; i++)
    {
      Vold[i] = Vnew[i];
      rold[i] = rnew[i];
    }
  }
}
```

Чтобы не загромождать код, были введены функции **`F_v()`**, **`F_m()`**, **`F_h()`**, **`F_n()`** и **`F_r()`**, в которых реализованы вычисления правых частей из (1)–(3).

Код функций вынесен в отдельный файл **functions.cpp**, а объявления прототипов функций и необходимые константы – в заголовочный файл **functions.h**.

Код этих файлов приводим без дальнейших комментариев, предоставляя читателям убедиться в его соответствии описанию из § 2.

```
// functions.h
#ifndef __FUNCTIONS_H__
#define __FUNCTIONS_H__

#include "math.h"

extern const int N;
const double V_na=-115.0, V_k=12.0, V_l=-10.613,
  g_na=120.0, g_k=36.0, g_l=0.3;
const double a=0.5, b=0.1, E_syn =-10.0;
const double a_ex=2,b_ex=1, E_ex=70.0;

double F_v(int num, double *g, double V, double *r,
  double m, double h, double n, double I_ext);
double F_m(double V, double m);
double F_h(double V, double h);
double F_n(double V, double n);
double F_r(double V, double r);
double F_gi(double r_pre, double r_post, double g);

#endif
```

```
// functions.cpp
#include "functions.h"

double F_v(int num, double *g,double V, double *r,
  double m, double h, double n, double I_ext)
{
  double I_k = g_k * n * n * n * n * (V + V_k);
  double I_na = g_na * m * m * m * h * (V + V_na);
  double I_leak = g_l * (V + V_l);
  double I_syn = 0.0;
  for (int j = 0; j < N; j++)
    I_syn = I_syn - r[j] * g[j] * (E_syn - V);
  double res = -(I_k + I_na + I_leak + I_syn + I_ext);

  return res;
}

double F_m(double V, double m)
{
  double alpha =
```

```
    (0.1 * (-V + 25.0)) / (exp((-V + 25.0) / 10.0) - 1.0);
  double beta = 4.0 * exp(-V / 18.0);
  double res = alpha * (1.0 - m) - beta * m;

  return res;
}

double F_h(double V,double h)
{
  double alpha = 0.07 * exp(-V / 20.0);
  double beta = 1.0 / (exp((-V + 30.0) / 10.0) + 1.0);
  double res = alpha * (1 - h) - beta * h;

  return res;
}

double F_n(double V,double n)
{
  double alpha =
    (0.01 * (-V + 10.0)) / (exp((-V + 10.0) / 10.0) - 1.0);
  double beta = 0.125 * exp(-V / 80.0);
  double res = alpha * (1 - n) - beta * n;

  return res;
}

double F_r(double V, double r)
{
  double res;
  double T = 1.0 / (1 + exp(-(V - 20.0) / 2.0));
  res = a * T * (1 - r) - b * r;

  return res;
}

double F_gi(double r_pre, double r_post, double g)
{
  return 0.0;
}
```

Отметим также, что на каждой итерации внутреннего цикла по нейронам инициализируется значение очередного параметра **`I_ext`** с использованием случайного «довеска» из диапазона от 0 до 1. Этот факт окажет в дальнейшем определенное влияние на разработку параллельной версии.

Осталось реализовать последнюю функцию – печать результатов в файл.

```
void Print()
{
```

```
  FILE *f = fopen("test_s.csv", "w+");
  for (int i = 0; i < N; i++)
  {
    fprintf(f, "%lf;%lf;%lf;%lf;%lf;\n", Vold[i], m[i],
      h[i], n[i], rold[i]);
  }
  fclose(f);
}
```

Соберите проект и запустите его на выполнение, замерьте время работы. На тестовой инфраструктуре, описанной в § 1.3, авторами получено время 17,174 секунды.

4.2. Параллельная версия. Подход 1

Попытаемся распараллелить описанную выше реализацию, используя возможности технологии OpenMP. На идейном уровне все просто – в силу независимости вычислений в цикле по нейронам все, что необходимо, – распределить его по потокам, добавив перед циклом следующую директиву.

```
#pragma omp parallel for num_threads(NumThreads)
```

Здесь NumThreads – число потоков. Значение этой переменной будем читать из командной строки,

```
  if (argc > 1)
    NumThreads = atoi(argv[1]);
```

а инициализируем ее единицей.

```
int NumThreads = 1;
```

Отметим, что с точки зрения работы с переменными полученная таким простым способом OpenMP-версия является корректной – единственная изменяемая переменная **`i`** будет локализована автоматически. Однако не все так просто.

Для проведения экспериментов создадим в рамках решения **ODE** новый проект с названием **02_OpenMP**. Повторите все действия, описанные в § 4.1, с той лишь разницей, что начать нужно с выбора решения **ODE** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_omp**.

После получения пустого файла **main_omp.cpp** скопируйте в него код из файла **main_s.cpp** проекта **01_Serial**. Добавьте указанную выше директиву и чтение значения переменной NumThreads из командной строки. Далее включите в настройках проекта поддержку OpenMP. В дереве **Configuration Properties** перейдите к разделу **C/C++→Language** и в поле **OpenMP Support** справа выберите вариант: **Generate Parallel Code (/openmp, equiv. to /Qopenmp)**.

Соберите проект, запустите и убедитесь, что результаты работы отличаются от тех, что были получены в последовательной версии, но при этом не меняются от запуска к запуску.

В чем может быть дело? Гонок данных в программе нет – все изменяемые переменные меняются потоками независимо. Тем не менее, в программе, конечно же, содержится ошибка. Догадаться о том, чем она вызвана, весьма непросто – требуются знания об устройстве генераторов псевдослучайных чисел (ПСЧ) и особенностях их применения в многопоточных программах. Данный вопрос заслуживает отдельного рассмотрения и подробно изучается в главе «Методы Монте-Карло» и соответствующей лабораторной работе. Рассмотрим некоторые моменты, необходимые для понимания текущего материала.

Генератор псевдослучайных чисел представляет собой некоторый алгоритм, цель которого – по возможности наилучшим образом имитировать заданное распределение (в данном случае речь идет о равномерном распределении). Один из таких алгоритмов реализован в стандартных библиотеках языка С в виде функции **`rand()`.** Начало последовательности ПСЧ определяется вызовом функции **`srand(seed)`**. Далее каждый вызов функции **`rand()`** порождает очередное число из последовательности ПСЧ, порожденной числом **`seed`.** Возможный способ реализовать такую функциональность – хранить в некоторой глобальной переменной текущее псевдослучайное число и создавать на его основе следующее. Однако этот способ в случае использования функции **`rand()`** в многопоточной программе немедленно приведет к гонке данных по «некоторой глобальной переменной». Чтобы этого избежать, функция **`rand()`** реализуется так, что каждый поток использует собственное инициализирующее значение (то, что мы задаем в вызове **`srand()`**), то есть, фактически в многопоточной программе мы имеем столько генераторов, сколько потоков. И, конечно, каждый новый поток использует для инициализации то же значение, что и главный поток программы. Таким образом, в нашем случае генераторы в каждом потоке выдают одну и ту же последовательность чисел, в чем нетрудно убедиться, выведя их на печать или в файл. Как результат, вместо N = 1000 разных чисел, как это было в последовательном случае, мы имеем при двух потоках два раза по 500 одинаковых чисел.

Теория и практика использования генераторов ПСЧ в параллельных программах достаточно разработана. Вначале рассмотрим простейший вариант, который состоит в том, что собственно генерация выполняется только одним потоком. Этот вариант мы реализуем сейчас, а обсуждение других способов отложим до следующего раздела.

Итак, теперь мы не можем выполнять инициализацию элементов массива **`I_ext`** внутри цикла по нейронам. Реализуем функцию **`GenIExt()`**, в которой будем заполнять все элементы **`I_ext()`**

```
void GenIExt()
{
  int i;
  for (i = 0; i < N; i++)
    I_ext[i] = -10.0 - (double)rand() / ((double)RAND_MAX);
}
```

и вызовем ее перед циклом по нейронам, то есть перед параллельной областью.

```
  for (count = 0; count < Time; count++)
  {
    GenIExt();
#pragma omp parallel for num_threads(NumThreads)
    for (i = 0; i < N; i++)
    {
      ...
```

Соберите проект и убедитесь, что результаты совпадают с исходной последовательной версией.

Итак, корректная параллельная версия получена. При этом генерация ПСЧ осуществляется в последовательном режиме. Насколько это критично, покажет эксперимент. Дело в том, что в каждой конкретной задаче ситуация будет претерпевать изменения в зависимости от соотношения времени генерации чисел и остального расчета. Заметим, во многих задачах требуются не только равномерные, но и более сложные распределения, что соответствующим образом влияет на рост затрат времени на генерацию ПСЧ. Существует достаточно приложений, в которых последовательная генерация ПСЧ сразу для всех потоков будет существенно ограничивать масштабируемость.

Приступим к экспериментам по оценке ускорения. На тестовой инфраструктуре, описанной в § 1.3, авторами получены следующие результаты (время в секундах).

Таблица 1. Сравнение последовательной и OpenMP версий

Число потоков	T_{Serial}	T_{OpenMP}	Speedup
1	17,174	15,722	-
2	-	7,924	1,98
4	-	4,399	3,57
8	-	2,683	5,86

Как видим, на восьми потоках ускорение не достигает 6.

Попробуем улучшить ситуацию, выполнив параллельную генерацию ПСЧ. Это может быть сделано разными способами (см. главу «Методы Монте-Карло» и лабораторную работу к ней), здесь же мы используем следующий подход. Нам необходима возможность получать случайные числа из одной и той же последовательности, начиная с разных номеров. Например, при числе нейронов 1000 и двух потоках первый поток «должен уметь получить» первые 500 случайных чисел, а второй – остальные, то есть числа последовательности, начиная с номера 501. И такая возможность для ряда алгоритмов существует! Значительная часть алгоритмов генерации ПСЧ дает возможность пропустить некоторое указанное количество членов последовательности, инициализированной определенным **`seed`**, то есть, фактически, перейти сразу к числу с выбранным номером.

Генератор **`rand()`** эту возможность не поддерживает, поэтому используем библиотеку **MKL**, содержащую качественные реализации генераторов ПСЧ, в частности ориентированных на параллельные вычисления.

4.3. Параллельная версия. Подход 2

Рассмотрим использование генераторов в той степени, в которой это требуется в данной работе (см. подробное описание в **mklman.pdf** [13] и **vslnotes.pdf** [14]). Итак, нам необходимо генерировать равномерно распределенные псевдослучайные числа. Для этого будем использовать один из базовых[1] генераторов библиотеки MKL – MCG59, характеризующийся достаточно большим периодом и допускающим использование в параллельных вычислениях[2].

Во-первых, нам понадобится служебная переменная:

```
VSLStreamStatePtr stream;
```

Во-вторых, необходимо инициализировать базовый генератор. Сделать это можно, используя следующую функцию:

```
vslNewStream(&stream, VSL_BRNG_MCG59, seed);
```

Константа **`VSL_BRNG_MCG59`** задает тип базового генератора (алгоритм получения псевдослучайных чисел – MCG59), а параметр **`seed`** – начальное значение, которым инициализируется генератор. Алгоритм получения псевдослучайных чисел является детерминированным, последовательность чисел определяется начальным значением.

---

[1] Здесь и далее «базовый генератор» – генератор равномерного распределения.

[2] Заметим, что MCG59 – известный, достаточно хорошо себя зарекомендовавший, но не единственный базовый генератор для параллельных вычислений в системах с общей памятью. В частности, с теми же целями может быть использован генератор MT2203 (см. подробнее vslnotes.pdf).

Равномерно распределенные ПСЧ будем получать, вызывая функцию **`vdRngUniform()`**, передавая в нее в качестве параметров константу, описывающую метод, служебную переменную **`stream`**, объявленную ранее, размер буфера, в который будут возвращены ПСЧ (генератор позволяет получать несколько членов последовательности ПСЧ за один вызов), сам буфер и параметры распределения (в данном случае диапазон).

По окончании вычислений память, выделенная нами для работы генератора, должна быть освобождена. Используем функцию **`vslDeleteStream()`**.

Теперь все готово для реализации второй параллельной версии. Но прежде создадим в рамках решения **ODE** новый проект с названием **03_MCG59**. Повторите все действия, описанные в § 4.1, с той лишь разницей, что начать нужно с выбора решения **ODE** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_mcg59**.

После получения пустого файла **main_mcg59.cpp** скопируйте в него код из файла **main_omp.cpp** проекта **02_OpenMP**. Следующим шагом подключим библиотеку **MKL**. Откройте свойства проекта (либо через команду **Properties** контекстного меню, либо через основное меню **Project**), в дереве **Configuration Properties** перейдите к разделу **Linker→Input** и в поле **Additional Dependencies** справа укажите список статических библиотек из **MKL**, необходимых для сборки проекта: **mkl_core.lib mkl_intel_c.lib mkl_sequential.lib**. Заметим, что подключение **mkl_sequential.lib** приводит к использованию последовательных версий функций MKL (альтернатива – **mkl_intel_thread.lib** – в данной работе не будет использована в связи с применением внешней схемы распараллеливания).

Итак, нам необходим способ получать случайные числа из одной и той же последовательности, начиная с разных номеров. В MKL эта возможность реализована посредством функции **`vslSkipAheadStream(stream, nskip)`**, которая для потока **`stream`** указывает количество чисел, которое необходимо пропустить, начиная с текущего псевдослучайного числа последовательности.

В параллельной версии функции **`Solve()`** на каждой итерации цикла по времени нам необходимо в каждом потоке получать N / NumThreads псевдослучайных чисел в соответствии с номерами потоков. При этом нужно правильно рассчитать, сколько чисел пропустить перед началом выполнения и сколько пропускать после каждой выполненной итерации.

Для получения номера потока воспользуемся функцией **`omp_get_thread_num()`**, вызов которой должен быть выполнен в параллельной области. Таким образом, мы уже не сможем использовать совмещенную директиву **`omp parallel for`**. Придется сначала создавать параллельную область, выполнять необходимые предварительные дей-

ствия и только потом распределять итерации цикла по нейронам между потоками. Следовательно, имеет смысл создание параллельной области поместить в самое начало функции, тем более что нам для каждого потока необходим свой генератор ПСЧ (но необходимо не забыть, что все они должны быть инициализированы одним и тем же **seed**).

Итак, код функции **Solve()** будет выглядеть следующим образом.

```
void Solve(int Time)
{
#pragma omp parallel num_threads(NumThreads)
  {
    int i, count;
    int num = omp_get_thread_num();
    VSLStreamStatePtr stream;
    vslNewStream(&stream, VSL_BRNG_MCG59, seed);
    vslSkipAheadStream(stream, N / NumThreads * num);

    // Цикл по времени
    for (count = 0; count < Time; count++)
    {
      vdRngUniform(VSL_METHOD_DUNIFORM_STD, stream,
        N / NumThreads, &I_ext[N / NumThreads * num],
        -11.0, -10.0);
      // Цикл по нейронам
#pragma omp for
      for (i = 0; i < N; i++)
      {
        Vnew[i] = Vold[i] + F_v(i, &G[i * N], Vold[i],
          rold, m[i], h[i], n[i], I_ext[i]) * dt;
        m[i] = m[i] + F_m(Vold[i], m[i]) * dt;
        h[i] = h[i] + F_h(Vold[i], h[i]) * dt;
        n[i] = n[i] + F_n(Vold[i], n[i]) * dt;
        rnew[i] = rold[i] + F_r(Vold[i], rold[i]) * dt;
      }
      // Копирование
#pragma omp single
      for (i = 0; i < N; i++)
      {
        Vold[i] = Vnew[i];
        rold[i] = rnew[i];
      }
      vslSkipAheadStream(stream, N - N / NumThreads);
    }
    delete [] I_ext;
    vslDeleteStream(&stream);
  }
}
```

В представленном коде полужирным начертанием выделены моменты, которые необходимо прокомментировать.

Первый отступ по последовательности случайных чисел каждый поток делает на **`N / NumThreads * num`** чисел от ее начала. После того, как выполнена одна итерация внешнего цикла по времени, каждый поток должен пропустить ту часть чисел, которую уже получили остальные потоки. Это число равно **`N - N / NumThreads`**. Именно его мы использовали во втором вызове функции **`vslSkipAheadStream()`**.

В функции **`vdRngUniform()`** необходимо указать буфер, куда будут размещены псевдослучайные числа. Поскольку общее количество чисел по прежнему равно *N*, мы можем использовать тот же массив **`I_ext`**, что и ранее. Нужно лишь указать, с какой позиции каждый поток должен размещать данные – **`&I_ext[N / NumThreads * num]`**.

И последнее: цикл копирования должен выполняться только одним потоком, поскольку его распараллеливание в силу крайней «легковесности» лишено смысла. Реализовать это можно разными способами, мы использовали самый простой – директиву **`omp single`**. Блок кода, к которому она применена, будет выполнен только одним (любым) потоком. И, что важно в данном случае, завершение директивы синхронизировано, то есть остальные потоки будут ждать окончания копирования, прежде чем перейти к следующей итерации цикла по времени.

Соберите получившийся проект и запустите на выполнение. Проведите эксперименты по оценке масштабируемости. Также убедитесь, что результаты запусков в несколько потоков совпадают с результатами запуска в один поток.

На тестовой инфраструктуре, описанной в § 1.3, авторами получены следующие результаты (время в секундах).

Таблица 2. Сравнение последовательной, OpenMP и MCG59 версий

Число потоков	T_{Serial}	T_{OpenMP}	Speedup	T_{MCG59}	Speedup
1	17,174	15,722	-	14,849	-
2	-	7,924	1,98	7,471	1,99
4	-	4,399	3,57	3,758	3,95
8	-	2,683	5,86	2,121	7,00

Как видим, параллельная версия, в которой случайные числа получаются в каждом потоке, не только демонстрирует существенно лучшую масштабируемость, но и в целом работает несколько быстрее. Последнее объясняется тем фактом, что реализация генератора в библиотеке MKL существенно превосходит по скорости стандартную функцию **`rand()`**.

5. Содержательная интерпретация

Содержательную интерпретацию результатов приведем для измененного способа инициализации, демонстрирующего эффекты при отсутствии

```
for(int i = 0; i < N; i++)
{
  I_ext[i] =
    -11.0 - 5.0 * (double)rand() / ((double)RAND_MAX);
  r[0][i] = 0.0;
  m[0][i] = (double)rand() / ((double)RAND_MAX);
  n[0][i] = (double)rand() / ((double)RAND_MAX);
  h[0][i] = (double)rand() / ((double)RAND_MAX);
  for(int j = 0; j < N; j++)
  {
    G[i * N + j]=0.0;
  }
  V[0][i] =
    0.0 + 50.0 * (double)rand() / ((double)RAND_MAX);
}
```

```
Solve(10000); // без связей
```

или наличии связей между нейронами.

```
for(int i = 0; i < N; i++)
{
  I_ext[i] =
    -11.0 - 5.0 * (double)rand() / ((double)RAND_MAX);
  r[0][i] = 0.0;
  for(int j = 0; j < N; j++)
  {
    G[i * N + j] =
      (5.0 + (double)rand() / ((double)RAND_MAX))
      / ((double)N);
  }
}
```

```
Solve(20000); // со связями
```

В данном численном эксперименте исследуется динамика сети биологически релевантых моделей нейронной активности (модель Ходжкина-Хаксли) в зависимости от силы синаптической связи (значений коэффициентов G_{ij}) между нейронами. На рис. 1 представлена пространственно-временная диаграмма, иллюстрирующая динамику части нейронной сети (первые 100 элементов из общего количества N = 1000 элементов). Цветом

отображается значения трансмембранного потенциала V_i. Красные полосы обозначают резкие всплески потенциала, что соответствует генерации потенциала действия (нервного импульса). Первые 100 мс нейроны не связаны между собой ($G_{ij} = 0$). В этот промежуток времени сеть демонстрирует асинхронную колебательную активность (генерацию несинхронизованных потенциалов действия – нервных импульсов). Такое поведение возникает из-за того, что каждая клетка имеет свою собственную, отличную от большинства других нейронов, частоту колебаний вследствие неоднородного распределения параметра I_i^{ext} по ансамблю. При включении связи (после первых 100 мс $G_{ij} \neq 0$) динамика сети существенно меняется. В результате взаимодействия в сети устанавливается режим синхронных колебаний, при этом все элементы делятся на три неравные группы: первая группа неосциллирующих нейронов (например, нейрон с номером 1 на рис. 1), и две группы, образующие два разных кластера синхронно осциллирующих элементов (например, элементы с номерами 30, 33, 70 из первого кластера и 28, 63, 66 из второго). Также можно видеть, что нейроны из разных кластеров генерируют импульсы в противофазе (сдвиг фазы колебаний близок к π, см. реализации трансмембранного потенциала на рис. 2). Таким образом, несмотря на исходную неоднородность и различную собственную частоту колебаний нейронов, синаптическое взаимодействие привело к тому, что часть нейронов синхронизировалась и выработала единый колебательный ритм.

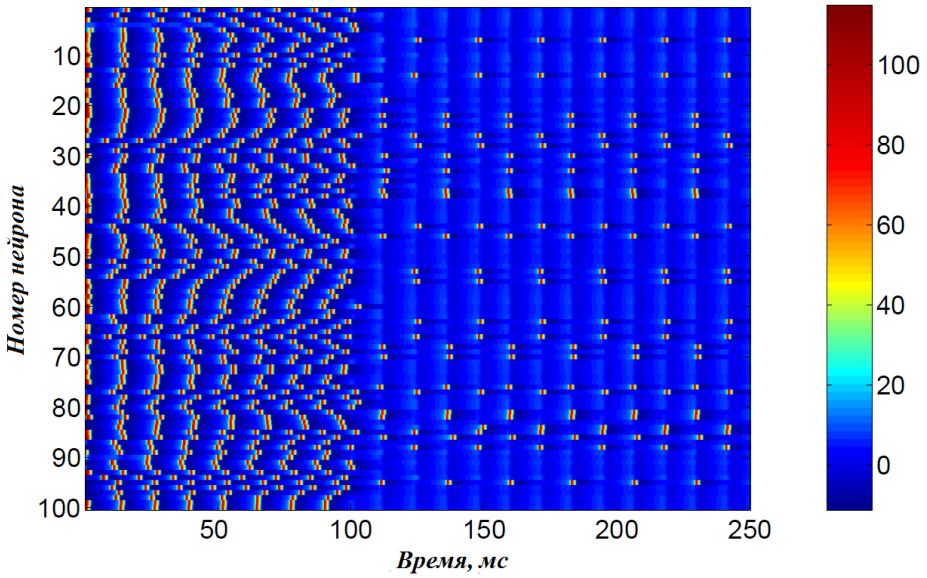

Рис. 1. Пространственно-временная диаграмма, иллюстрирующая динамику сети из $N = 1000$ элементов

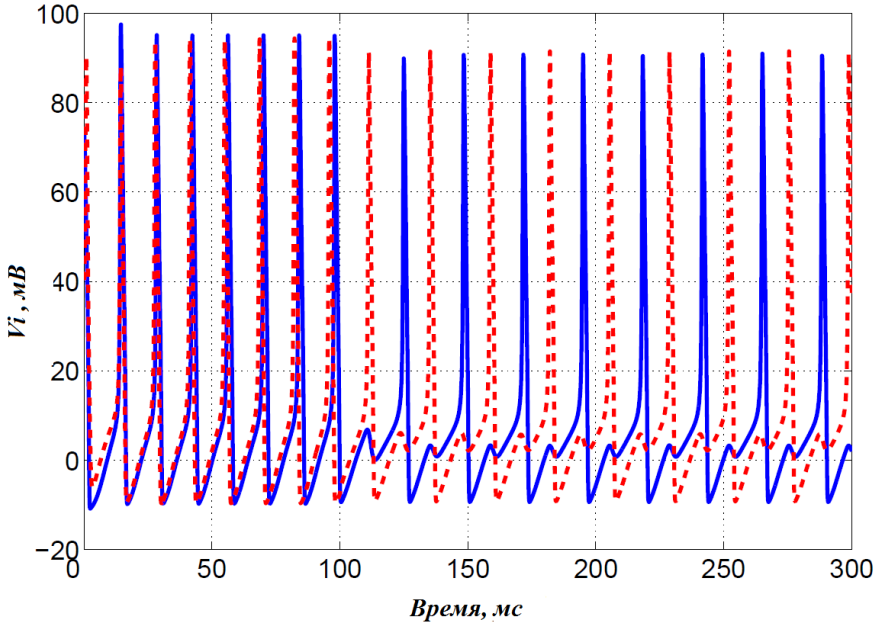

Рис. 2. Реализации значений трансмембранного потенциала для двух элементов из разных кластеров

6. Дополнительные задания

1. Самостоятельно реализуйте генераторы равномерного распределения MCG31 и MCG59 по описанию в [11, 14], а также технику Skip-ahead для их применения в параллельных вычислениях. Убедитесь в корректности работы генераторов.
2. Разработайте параллельную реализацию с использованием Intel Cilk Plus. Проведите вычислительные эксперименты. Проанализируйте корректность работы и полученное ускорение.
3. Разработайте параллельную реализацию с использованием Intel TBB. Проведите вычислительные эксперименты. Проанализируйте корректность работы и полученное ускорение.
4. Используйте Intel Amplifier XE для выявления наиболее затратных по времени участков кода. Обдумайте возможности по оптимизации (указание: один из ключевых циклов не векторизуется компилятором; тем не менее, он может быть векторизован, будучи развернут вручную).

7. Литература

7.1. Использованные источники информации

1. Rabinovich M.I., Varona P., Selverston A.I., Abarbanel H.D.I. Dynamical Principles in Neuroscience // Review of Modern Physics, vol. 78(4), 1213(53), 2006.
2. Николлс Дж., Мартин Р., Валлас Б., Фукс П.. От нейрона к мозгу. – М.:Издательство ЛКИ, 2008.
3. Izhikevich E.M. Dynamical systems in neuroscience: geometry of excitability and bursting. – MIT Press, 2006.
4. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. – М.: Наука, 1987.
5. Тихонов А. Н., Васильева А. Б., Свешников А. Г. Дифференциальные уравнения. — М.: Наука, 1985.
6. Самарский А.А., Гулин А.В. Численные методы. – М.: Наука, 1989.
7. Хайрер Э., Нёрсетт С., Ваннер Г. Решение обыкновенных дифференциальных уравнений. Нежесткие задачи. – М.: Мир, 1990.
8. P.J. van der Houwen, E. Messina. Parallel Adams methods // Journal of Computational and Applied Mathematics, vol. 101, 1999. pp. 153-165.
9. P. Amodio, L. Brugnano. Parallel solution in time of ODEs: some achievements and perspectives // Applied Numerical Mathematics, vol. 59, 2009. pp. 424-435.
10. M. Korch, T. Rauber. Parallel Implementation of Runge-Kutta Integrators with Low Storage Requirements. H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, 2009. pp. 785-796.

7.2. Дополнительная литература

11. Кнут Д. Искусство программирования, том 2. Получисленные методы. – 3-е изд. – М.: Вильямс, 2007. – 832 С.

7.3. Ресурсы сети Интернет

12. http://math.ucsd.edu/~williams/courses/sde.html
13. Intel Math Kernel Library Reference Manual.

[http://software.intel.com/sites/products/documentation/hpc/mkl/mklman.pdf].

14. Intel Vector Statistical Library Notes.

[http://software.intel.com/sites/products/documentation/hpc/mkl/vsl/vslnotes.pdf]