Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Робототехники и комплексной автоматизации

КАФЕДРА Системы автоматизированного проектирования (РК-6)

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Студент	Никифорова Ирина Андреевна
Группа	РК6-61б
Тип задания	лабораторная работа
Тема лабораторной работы	Программирование средствами MPI

Студент _____ **Никифорова И. А.**
подпись, дата *фамилия, и.о.*

Преподаватель _____ **Федорук В.Г.**
подпись, дата *фамилия, и.о.*

Оценка _____

Москва, 2019 г.

Оглавление

Задание на лабораторную работу

Разработать средствами MPI параллельную программу моделирования распространения электрических сигналов в линейной цепочке RC-элементов (рис. 1). Метод формирования математической модели - узловой. Метод численного интегрирования - явный Эйлера. Внешнее воздействие - источники тока и напряжения. Количество (кратное 8) элементов в сетке, временной интервал моделирования - параметры программы. Программа должна демонстрировать ускорение по сравнению с последовательным вариантом. Предусмотреть визуализацию результатов посредством утилиты gnuplot.

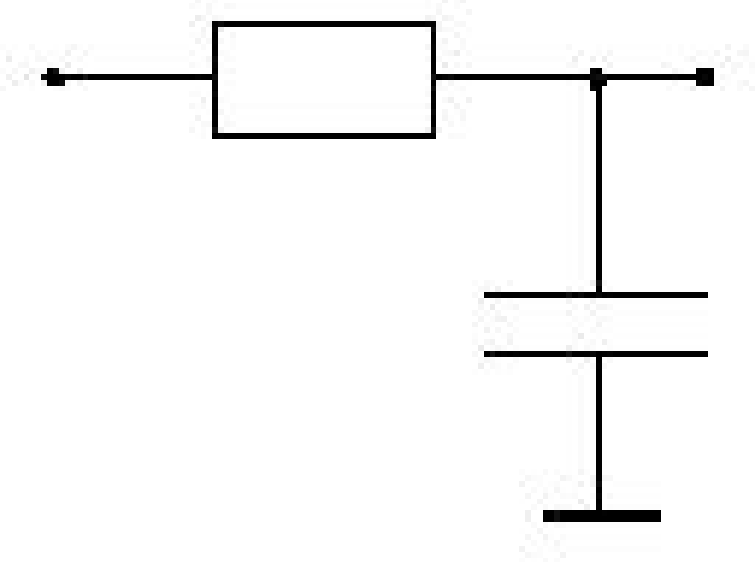

Рис. 1. Схема одной части RC-цепочки.

Теоретическая основа разработанной программы

Для формирования математической модели задачи был использован узловой метод. Он подразумевает использование второго закона Кирхгофа (1).

$$I_{Rл} - I_{Rn} - I_C = 0 \quad (1)$$

В формуле (1) $I_{Rл}$ - ток через левое от узла сопротивление, I_{Rn} - через правое, I_c - ток через ёмкость, присоединенную к узлу снизу. Они выражаются с помощью формул (2), (3) и (4) соответственно. В данных формулах U_i - электрический потенциал узла с номером *i.* Узлы нумеруются от нуля слева направо до *N*, где *N* - количество *RC*-элементов в схеме.

$$I_{Rл} = (U_{i-1} - U_i) / R \quad (2)$$

$$I_{Rn} = (U_i - U_{i+1}) / R \quad (3)$$

$$I_C = C \cdot dU_i / dt \quad (4)$$

Таким образом, уравнение баланса токов в узле (1) при учете уравнений (2), (3) и (4) примет вид (5).

$$(U_{i-1} - U_i) / R - (U_i - U_{i+1}) / R - C \cdot dU_i / dt = 0 \quad (5)$$

Для вычисления производной был применен метод для численного интегрирования ОДУ - явный метод Эйлера. Он подразумевает использование значения в узле на следующем шаге интегрирования. Таким образом, при использовании данного метода уравнение (5) преобразуется в уравнение (6).

$$(U^n_{i-1} - U^n_i) / R - (U^n_i - U^n_{i+1}) / R - C \cdot (U^{n+1}_i - U^n_i) / h_t = 0 \quad (6)$$

В формуле (6) величина h_t - представляет собой шаг по времени для численного интегрирования.

При выражении неизвестной U^{n+1}_i из уравнения (6) было получено уравнение (7).

$$U^{n+1}_i = h_t / (R \cdot C) \cdot (U^n_{i-1} - 2 \cdot U^n_i + U^n_{i+1}) + U^n_i \qquad (7)$$

Таким образом, исходя из формулы (7) на каждом шаге по времени, можно получить значение напряжения в узле, используя данные о значениях в этом же и соседних с ним узлах с предыдущего временного слоя.

Для корректного решения задачи необходимо также задать начальные и граничные условия.

Описание структуры программы и реализованных способов взаимодействия процессов

Идея реализации программы состоит в том, чтобы разделить вычисления напряжения U_i в узлах между несколькими процессами. Таким образом, время расчета значительно сократится. Делить данные было решено на количество процессов, являющееся делителем восьми, т.к. можно запустить максимально 8 процессов на кластере (с максимальным выигрышем по времени). Для того, чтобы узлы правильно распределялись по процессам, необходимо, чтобы их общее количество, исключая граничные, было кратно восьми. Схемы разделения данных по потокам на примере десяти узлов представлены на рисунках 2 - 5. На данных рисунках красным отмечены граничные узлы (они не входят в основной расчет, ими отдельно занимается root-процесс), зеленым отмечены вычисляемые узлы, сверху в шестиугольнике указан номер процесса, рассчитывающего данные узлы.

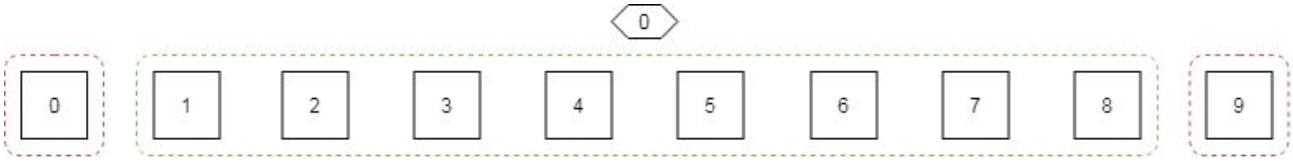

Рис. 2. Разделение расчета для десяти узлов на один процесс.

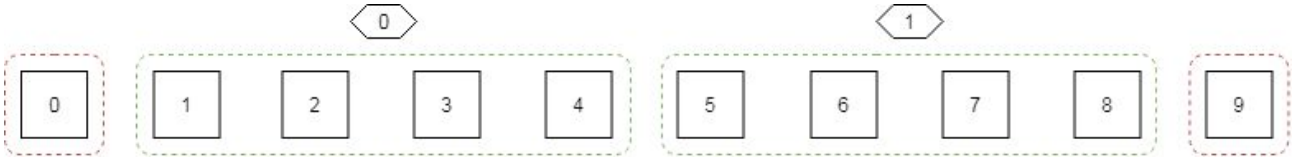

Рис. 3. Разделение расчета для десяти узлов на два процесса.

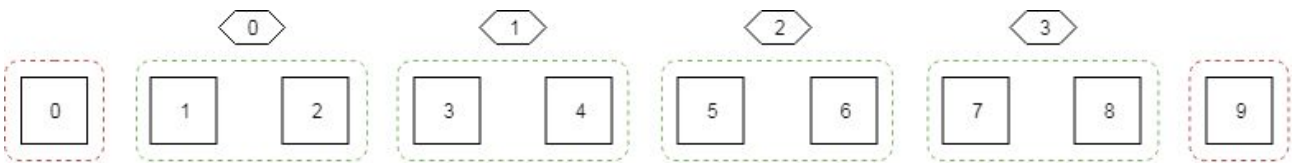

Рис. 4. Разделение расчета для десяти узлов на четыре процесса.

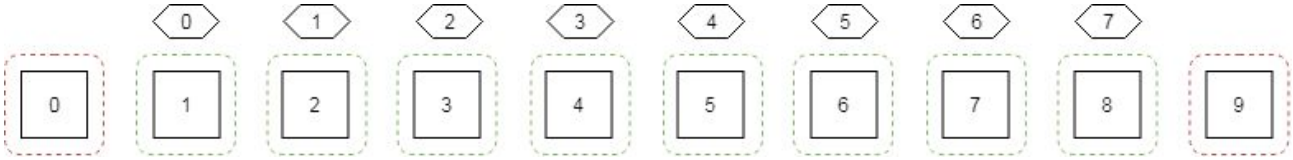

Рис. 5. Разделение расчета для десяти узлов на восемь процессов.

После того, как каждый процесс рассчитает свои узлы, ему необходимо отправить их в общую матрицу U в root-процессе, а также соседним процессам, так как им нужно знать эти значения для расчета своих узлов на следующем шаге. Схема общения процессов представлена на рисунке 6. Для удобства восприятия root-процесс был представлен отдельно от нулевого процесса, хотя по факту это один и тот же процесс, выполняющий сразу две функции: рассчитывающий узлы, как и все остальные процессы, и собирающий информацию, как управляющий.

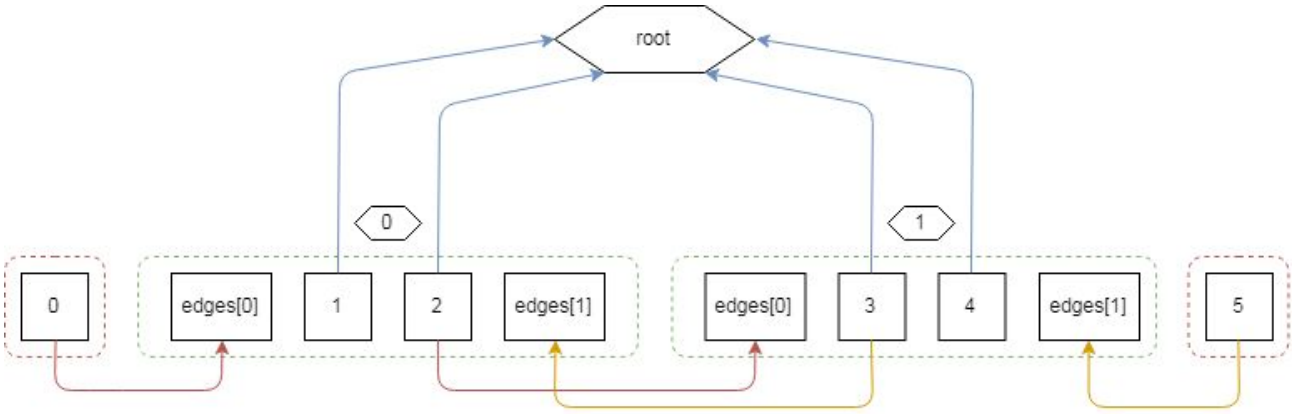

Рис. 6. Передача данных между процессами. Синим цветом вверху обозначена передача root-процессу (сбор информации), передаются все узлы. Красным цветом обозначена передача процессу его левого граничного узла. Желтым - правого.

На рисунке 6 также можно отметить, что нулевой и первый процесс принимают и передают данные отлично от других процессов. Нулевой процесс в качестве левой границы должен принять граничный узел для всей схемы (

узел 0 в общей нумерации). Он также не должен передавать свой узел (узел 1) никакому процессу. Аналогично и последний процесс.

Реализация описанного выше алгоритма решения задачи составила два файла.

Первый, заголовочный файл, main.h содержит объявления констант (таких как сопротивление резисторов или емкость конденсаторов, шаг интегрирования и другие).

Второй файл, main.cpp, содержит три функции. Первая из них, функция *make_script* отвечает за создание скрипта gnuplot для отображения результатов работы в графической форме.

Функция *get_args* анализирует аргументы программы на правильность и помещает их в специальные глобальные переменные. На вход программе должны подаваться два аргумента: количество узлов в цепочке и время интегрирования. Нельзя забывать, что для корректной работы программы необходимо, чтобы количество узлов, исключая граничные было кратно восьми.

Основная работа программы сконцентрирована в функции *main*. Сначала там происходит запуск MPI с помощью функции *MPI_Init*. Далее программа узнает общее количество процессов с помощью *MPI_Comm_size* и номер своего процесса с помощью *MPI_Comm_rank.* Эти данные будут необходимы для корректного разделения вычислений по процессам и правильной связи процессов между собой. Далее в *main* создаются переменные, ипользуемые далее по коду. Они будут видны во всех процессах.

После этого идет часть, которая выполняется только root-процессом. Здесь открывается файл для записи данных и выделяется память для основной матрицы U, куда с каждого потока будут поступать рассчитанные значения. Также, чуть дальше root-процесс запускает таймер для отсчета времени работы

программы. В зависимости от того, какое конкретно время нужно измерить, запуск таймера можно переставить в другое место, как и снятие его показаний.

Следующим этапом каждый процесс, включая root, выделяет память под свой массив u_prev (значения потенциалов узлов на предыдущем шаге интегрирования) и u (на текущем). После этого, им необходимо узнать номера соседних с ними процессов.

Каждый процесс также инициализирует матрицу u_prev на данном этапе начальными условиями программы - U_0. Граничные условия - edges_prev - также задаются начальными для всех процессов, кроме первого и последнего - для них задаются реальные граничные условия всей системы. Массив edges_prev используется как граничные узлы на предыдущем этапе, а edges - на текущем.

После этого в программе идет основной цикл. В нем вычисления происходят по значениям переменной t, обозначающей момент времени. На каждой итерации к ней прибавляется h_t - шаг интегрирования. В самом цикле вычисляются значения потенциалов в узлах, принадлежащих выполняющему процессу. Они вычисляются с учетом заданных ранее edges_prev. Далее процесс отправляет и принимает массивы edges с помощью функций *MPI_Send* и *MPI_Recv*. Причем, здесь учитываются особенности первого и последнего процессов. После этого начинается процесс перехода к следующей итерации. Каждый процесс переносит свой массив edges в массив edges_prev и, после того, как все процессы закончат свои вычисление и подойдут к барьеру (это проверяет функция *MPI_Barrier*), отправляет root-процессу массив вычисленных значений u с помощью функции *MPI_Gather*. После этого root-процесс выводит полученные значения в файл, а каждый процесс перезаписывает u в u_prev для корректного продолжения вычислений.

После того, как цикл вычислений для заданных параметров пройден, root-процесс закрывает файл, выводит время вычислений, создает файл со скриптом для утилиты gnuplot и очищает память массивы U.

Каждый процесс дальше должен очистить свою память, где находились массивы u и u_prev и завершить работу с помощью функции *MPI_Finalize*.

Описание основных используемых структур данных

В программе использовались только массивы языка СИ.

Блок-схема программы

На рисунках 7 - 10 представлена блок-схема функции main написанной программы. Функция main составляет основную часть программы, поэтому другие функции описаны блок-схемой не были.

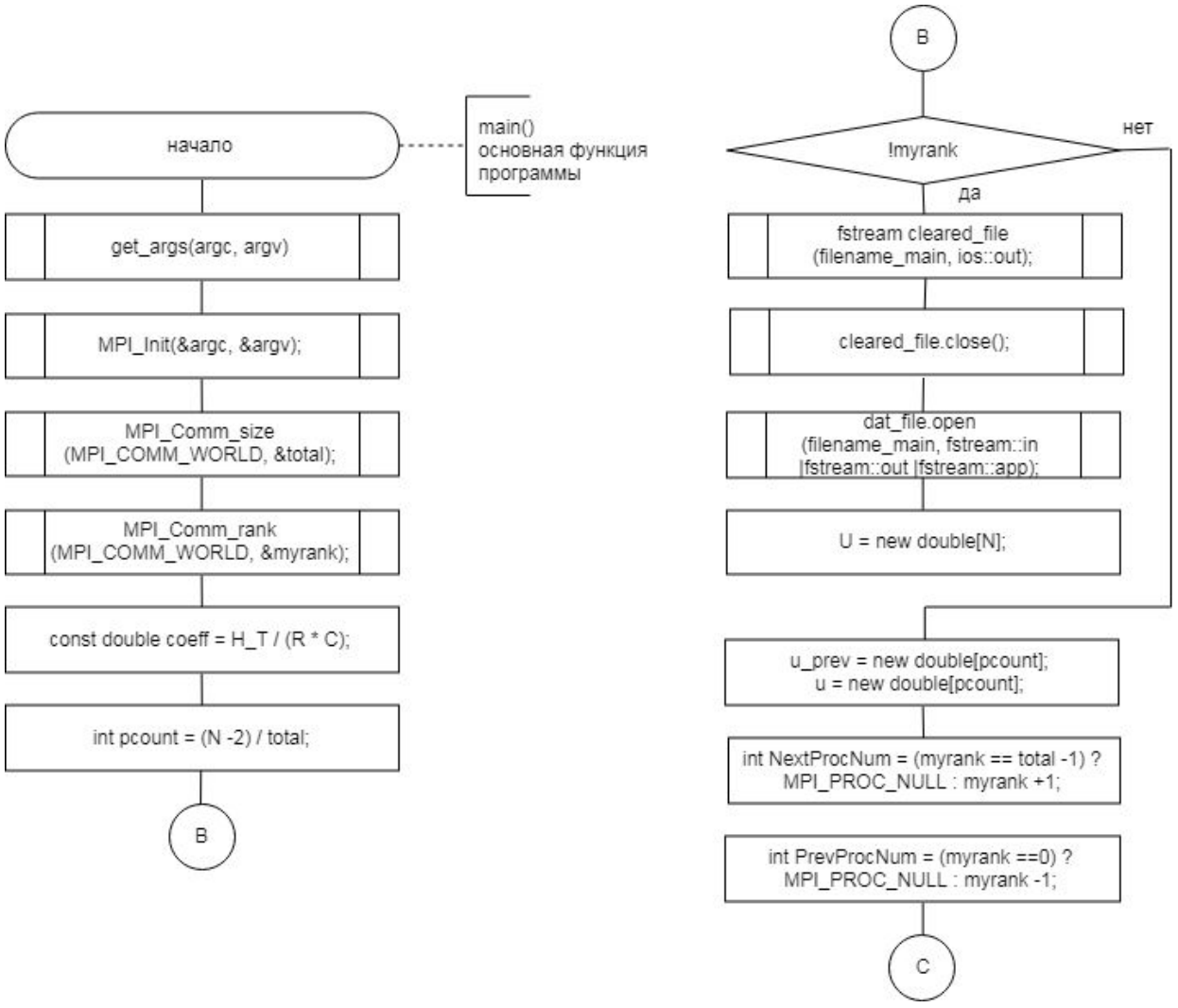

Рис. 7. Блок-схема функции main, часть 1

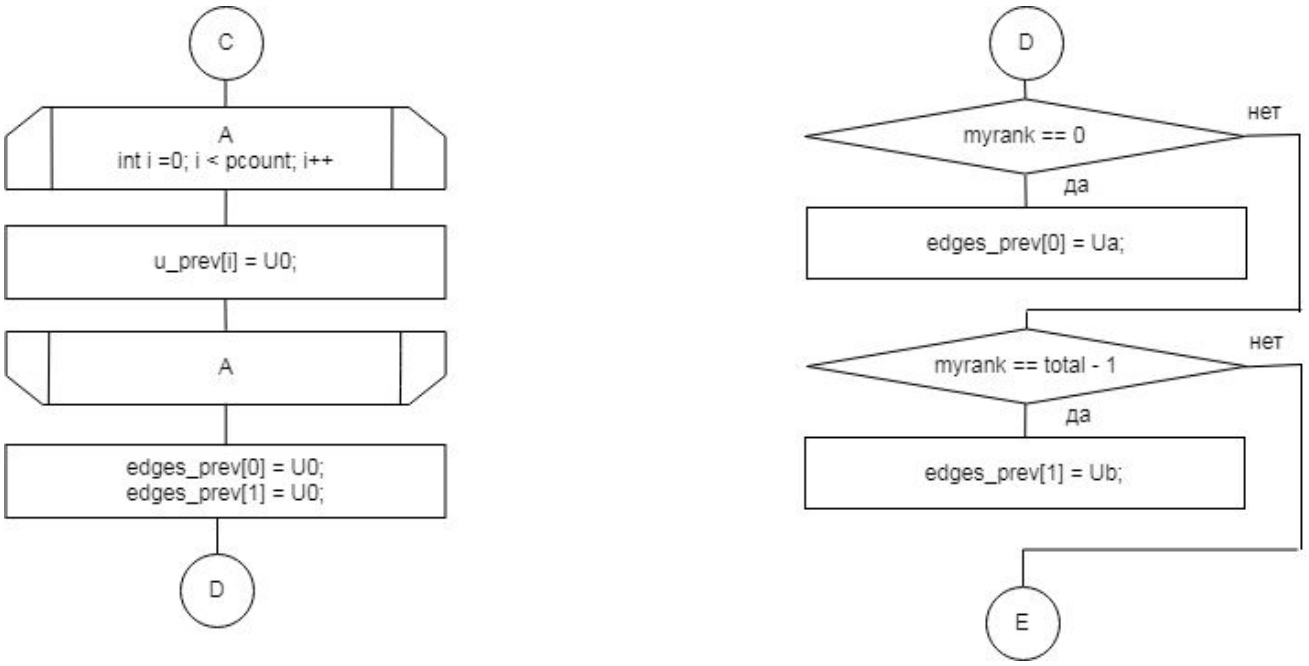

Рис. 8. Блок-схема функции main, часть 2

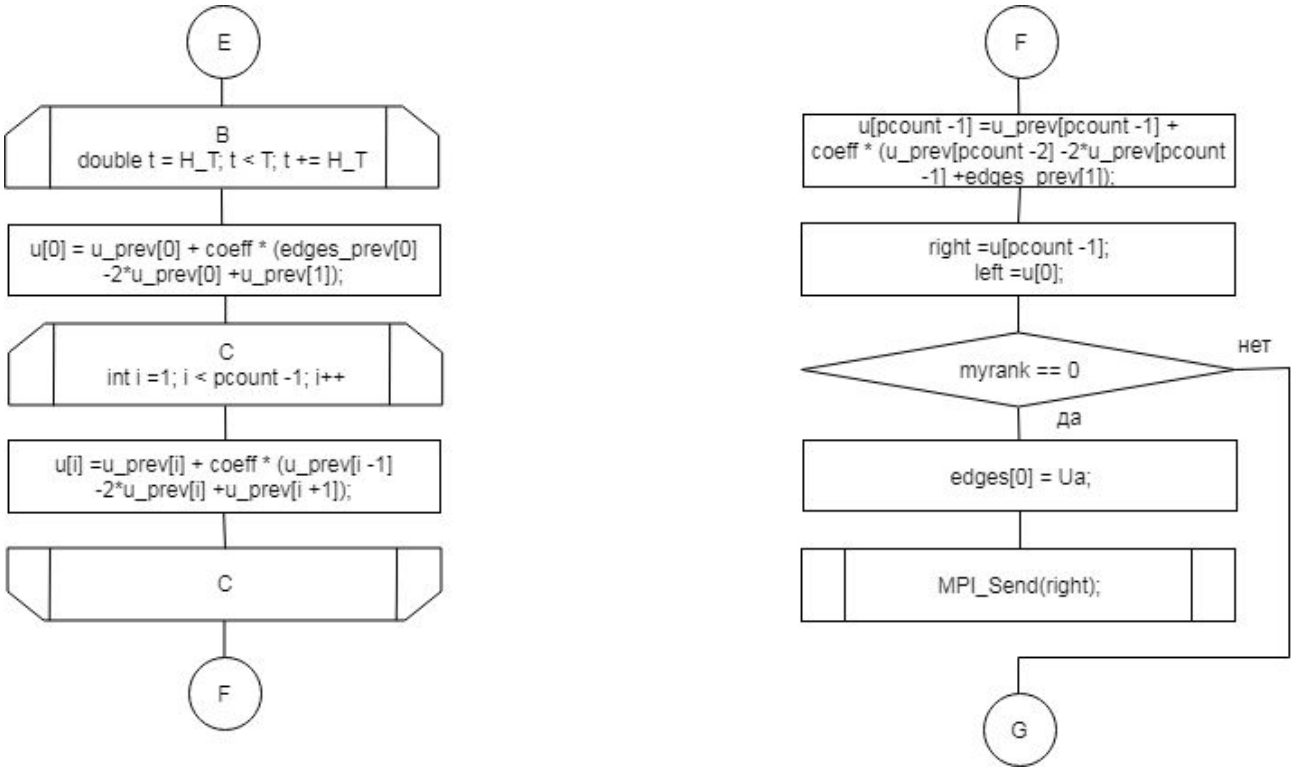

Рис. 9. Блок-схема функции main, часть 3

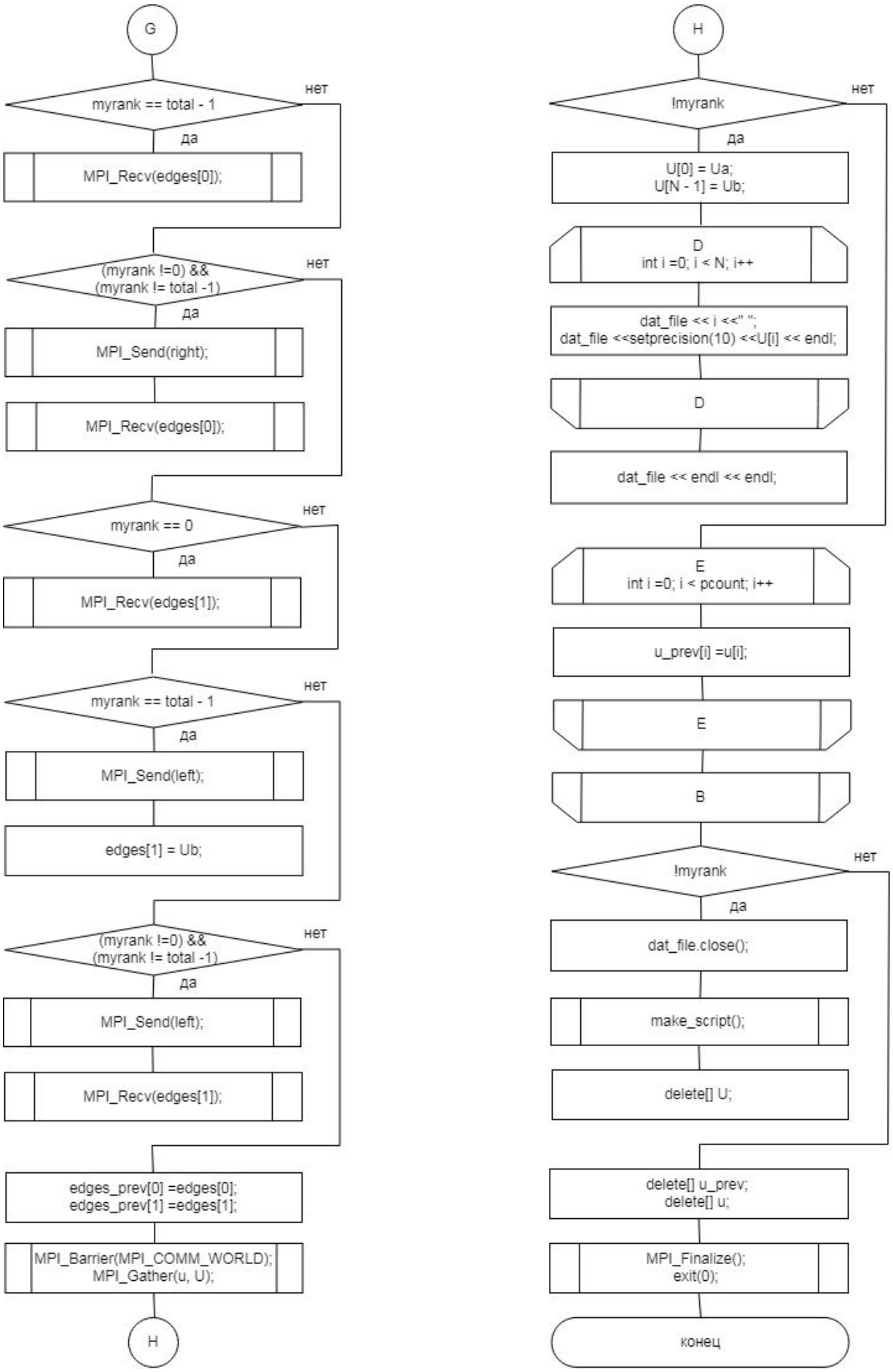

Рис. 10. Блок-схема функции main, часть 4

Примеры результатов работы программы

На рисунке 11 представлен график одного из состояний схемы, в листинге 1 можно увидеть вывод программы в файл.

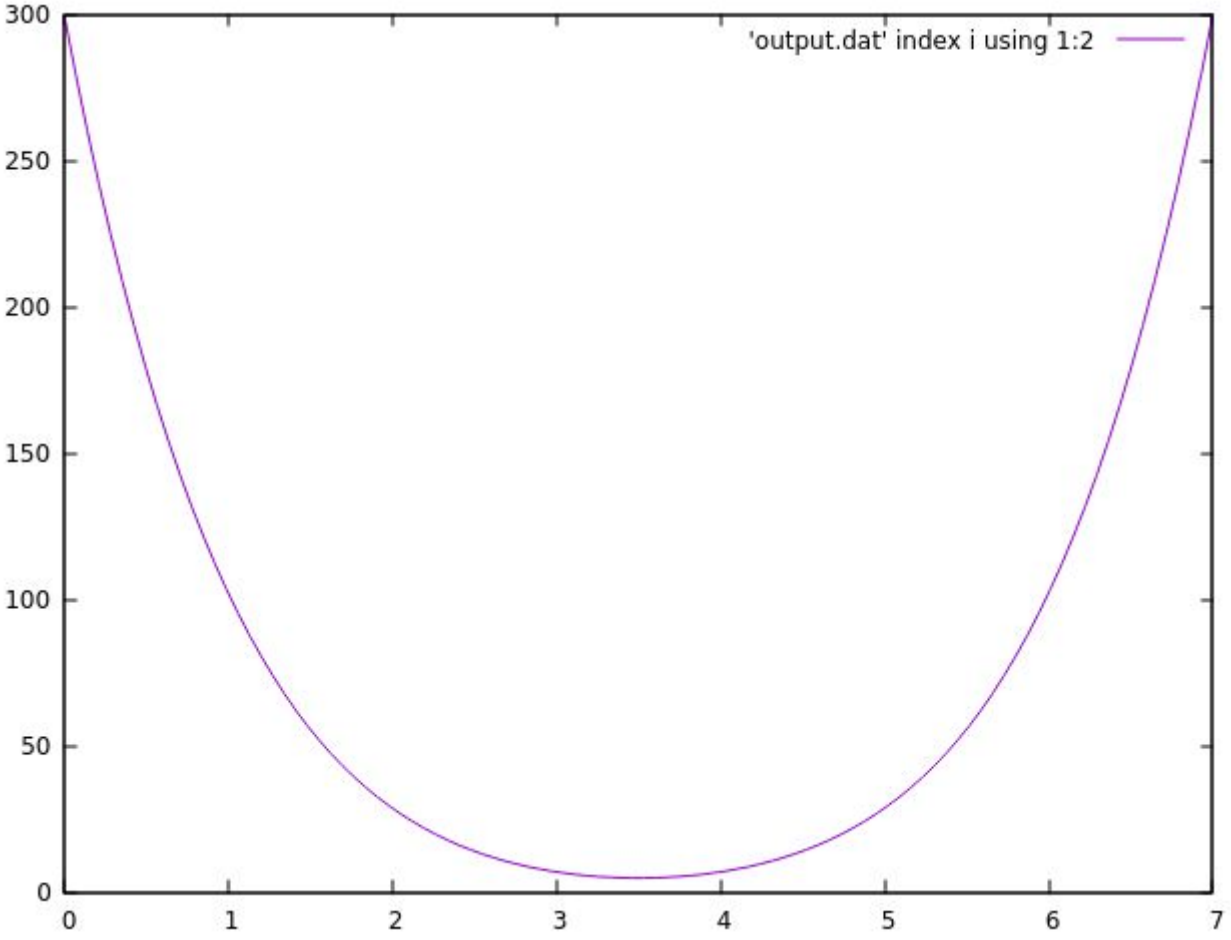

Рис. 11. Графический результат работы программы

Листинг 1. Текстовый результат первых пяти шагов интегрирования

```
0 300
1 0.003
2 0
3 0
4 0
5 0
6 0.003
7 300


0 300
1 0.00599994
2 3e-08
3 0
4 0
```

```
5 3e-08
6 0.00599994
7 300


0 300
1 0.008999820002
2 8.99988e-08
3 3e-13
4 3e-13
5 8.99988e-08
6 0.008999820002
7 300


0 300
1 0.01199964001
2 1.799952e-07
3 1.199985e-12
4 1.199985e-12
5 1.799952e-07
6 0.01199964001
7 300


0 300
1 0.01499940001
2 2.999880002e-07
3 2.999925001e-12
4 2.999925001e-12
5 2.999880002e-07
6 0.01499940001
7 300
```

Текст программы

В листинге 2 и 3 представлены тексты файлов программы.

Листинг 2. Файл main.h

```
#ifndef LAB4_MPI_MAIN_HPP
#define LAB4_MPI_MAIN_HPP

// значения сопротивления и емкости
#define R 1000
#define C 1e-6

// шаг интегрирования
#define H_T 0.00000001

//граничные условия
#define Ua 300
#define Ub 300
#define Ia 300
#define Ib 300

// начальные условия
#define U0 0

#endif  // LAB4_MPI_MAIN_HPP
```

Листинг 3. Файл main.cpp

```
#include <mpi.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <time.h>
#include "main.hpp"

using namespace std;

// возможность повторного входа в функцию разными процессами
#define _REENTRANT

// константы программы, определяемые
// из аргументов командной строки
long int N; // количество элементов
long double T; // интервал времени

// установка констант из аргументов программы
int get_args(int argc, char** argv) {
    if (argc < 3) {
        printf("Too few arguments\n");
        return -1;
    }

    char** endptr = NULL;
    N = strtol(argv[1], endptr, 10);
    if (endptr == &argv[1]) {
        printf("Num of elements is not a number\n");
        return -1;
    }

    if (((N - 2) % 8) != 0) {
        printf("Num of elements - 2 can't be divided by 8\n");
        return -1;
    }

    endptr = NULL;
    T = strtof(argv[2], endptr);
    if (endptr == &argv[2]) {
        printf("Time interval is not a number\n");
        return -1;
    }

    return 0;
}

// создание скрипта для отрисовки в gnuplot
int make_script() {
    // имя файла со скриптом для gnuplot
    const char* filename = "script.gnu";

    // очистка содержимого файла
    fstream cleared_file(filename, ios::out);
```

```
        cleared_file.close();

        // открытие файла, запись туда скрипта для gnuplot и закрытие
        fstream dat_file;
        dat_file.open(filename, fstream::in | fstream::out | fstream::app);

        dat_file << "set xrange[0:" << N - 1 << "]\n";
        dat_file << "set yrange[-200:200]\n";
        dat_file << "do for [i=0:" << (int)(T / (10 * H_T)) << "]{\n";
            dat_file << "plot 'output.dat' index i using 1:2 smooth
bezier\npause 0.1}\npause -1\n";

        dat_file.close();
    }


    int main(int argc, char** argv) {
        if (get_args(argc, argv) == -1) {
            return -1;
        }

        int myrank; // собственный идентификатор процесса
        int total;  // общее количество процессов в группе
        MPI_Status status; // статус при отправке и получении сообщений

        // запуск MPI
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &total);

        // чтобы каждый процесс узнал свой id
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

        // файл, куда будут записываться данные
        fstream dat_file;

        // массивы значение на текущем и на предыдущем шаге
        // глобальные (root) и локальные (для каждого процесса)
        double *U;
        double *u_prev, *u;

        // коэффициент, который будет использоваться в уравнениях
        const double coeff = H_T / (R * C);

        // расчет количества данных для одного процесса
        int pcount = (N - 2) / total;

        // подготовка исходных данных процессом root (id = 0)
        if (!myrank) {
            const char* filename_main = "output.dat";

            fstream cleared_file(filename_main, ios::out);
            cleared_file.close();

            dat_file.open(filename_main, fstream::in | fstream::out |
            fstream::app);

            // выделение памяти для массивов
```

```
            U = new double[N];
        }


        // выделение памяти для матриц u каждого процесса
        u_prev = new double[pcount];
        u = new double[pcount];

        // для измерения времени
        clock_t start_time, end_time, all_time;
        if (!myrank) {
            start_time = clock();
        }

        // все процессы узнают соседей и получают начальные условия
         int NextProcNum = (myrank == total - 1) ? MPI_PROC_NULL : myrank +
1;
         int PrevProcNum = (myrank == 0) ? MPI_PROC_NULL : myrank - 1;

        for (int i = 0; i < pcount; i++) {
            u_prev[i] = U0;
        }

        // граничные условия для начального состояния
        double edges_prev[2], edges[2], right, left;
        edges_prev[0] = U0;
        edges_prev[1] = U0;

        if (myrank == 0) {
            edges_prev[0] = Ua;
        }

        if (myrank == total - 1) {
            //edges_prev[1] = Ub;
             edges_prev[1] = H_T / C * ((u_prev[pcount - 2] + u_prev[pcount -
1]) / R + Ib) + u_prev[pcount - 1];

        }

        edges[0] = 0;
        edges[1] = 0;


        for (double t = H_T; t < T; t += H_T) {
            // каждый процесс считает свою часть
              u[0] = u_prev[0] + coeff * (edges_prev[0] - 2 * u_prev[0] +
u_prev[1]);
            for (int i = 1; i < pcount - 1; i++) {
                 u[i] = u_prev[i] + coeff * (u_prev[i - 1] - 2 * u_prev[i] +
u_prev[i + 1]);
            }
            u[pcount - 1] = u_prev[pcount - 1] +
                    coeff * (u_prev[pcount - 2] - 2 * u_prev[pcount - 1] +
edges_prev[1]);

            // потом отправляет и принимает edges[2]
            right = u[pcount - 1];
```

```
left = u[0];

// правая граница текущего процесса превращается
// в левую границу следующего
if (myrank == 0 ) {
    MPI_Send(
        (void*)(&right),
        1,
        MPI_DOUBLE,
        NextProcNum,
        1,
        MPI_COMM_WORLD
    );
    edges[0] = Ua;
}
if (myrank == total - 1) {
    MPI_Recv(
        (void*)(&edges[0]),
        1,
        MPI_DOUBLE,
        PrevProcNum,
        MPI_ANY_TAG,
        MPI_COMM_WORLD,
        &status
    );
}
if ((myrank != 0) && (myrank != total - 1)) {
    MPI_Send(
        (void*)(&right),
        1,
        MPI_DOUBLE,
        NextProcNum,
        1,
        MPI_COMM_WORLD
    );
    MPI_Recv(
        (void*)(&edges[0]),
        1,
        MPI_DOUBLE,
        PrevProcNum,
        MPI_ANY_TAG,
        MPI_COMM_WORLD,
        &status
    );
}

// левая граница текущего процесса превращается
// в правую границу предыдущего
if (myrank == 0 ) {
    MPI_Recv(
        (void*)(&edges[1]),
        1,
        MPI_DOUBLE,
        NextProcNum,
        MPI_ANY_TAG,
        MPI_COMM_WORLD,
        &status
```

```
            );
        }
        if (myrank == total - 1) {
            MPI_Send(
                (void*)(&left),
                1,
                MPI_DOUBLE,
                PrevProcNum,
                1,
                MPI_COMM_WORLD
            );
            //edges[1] = Ub;
             edges[1] = H_T / C * ((u_prev[pcount - 2] + u_prev[pcount -
1]) / R + Ib) + u_prev[pcount - 1];
        }
        if ((myrank != 0) && (myrank != total - 1)) {
            MPI_Send(
                (void*)(&left),
                1,
                MPI_DOUBLE,
                PrevProcNum,
                1,
                MPI_COMM_WORLD
            );
            MPI_Recv(
                (void*)(&edges[1]),
                1,
                MPI_DOUBLE,
                NextProcNum,
                MPI_ANY_TAG,
                MPI_COMM_WORLD,
                &status
            );
        }

        // перезаписывает edges_prev
        edges_prev[0] = edges[0];
        edges_prev[1] = edges[1];

        // отправляет руту, тот записывает в файл
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Gather(
            (void*)u,
            pcount,
            MPI_DOUBLE,
            (void*)(U + 1),
            pcount,
            MPI_DOUBLE,
            0,
            MPI_COMM_WORLD
        );

        // дополнение граничными условиями и запись в файл
        if (!myrank) {
            // граничные условия
            U[0] = Ua;
            //U[N - 1] = Ub;
```

```
                    U[N - 1] = edges[1] = H_T / C * ((u_prev[pcount - 2] +
u_prev[pcount - 1]) / R + Ib) + u_prev[pcount - 1];

                // запись значений в файл
                for (int i = 0; i < N; i++) {
                    dat_file << i << " ";
                    dat_file << setprecision(10) << U[i] << endl;
                }
                dat_file << endl << endl;
            }

            for (int i = 0; i < pcount; i++) {
                u_prev[i] = u[i];
            }
        }

        // завершение работ root-процесса
        if (!myrank) {
            dat_file.close();

            // завершение подсчета времени
            end_time = clock();
            all_time = (end_time - start_time) / CLOCKS_PER_SEC;
            cout << all_time << "s" << endl;

            // создание root-процессом скрипта для gnuplot
            make_script();

            // очистка памяти
            delete[] U;
        }

        // завершение работы
        delete[] u_prev;
        delete[] u;
        MPI_Finalize();
        exit(0);
    }
```