Министерство науки и высшего образования 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 Робототехника и комплексная автоматизация

КАФЕДРА Системы автоматизированного проектирования (РК-6)

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

по дисциплине: «Разработка программных систем»

Студент Гусаров Аркадий Андреевич

Группа РК6-63Б

Тип задания Лабораторная работа №4

Вариант лабораторной работы 10

Студент _____ **Гусаров А.А.**
подпись, дата *фамилия, и.о.*

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

Оценка _____

Москва, 2022 г.

ОГЛАВЛЕНИЕ

Текст задания

Разработать средствами MPI параллельную программу решения уравнения струны методом конечных разностей с использованием явной вычислительной схемы. Временной интервал моделирования и количество (кратное 8) узлов расчетной сетки - параметры программы.

Уравнение струны имеет следующий вид:

$$\frac{d^2z}{dt^2} = a^2 * \frac{d^2z}{dx^2} + f(x,t)$$

где *t* – время, *x* – пространственная координата, вдоль которой ориентирована струна, *z* – отклонение (малое) точки струны от положения покоя, *a* – фазовая скорость, *f(x, t)* – внешнее "силовое" воздействие на струну.

Предусмотреть возможность задания ненулевых начальных условий и ненулевого внешнего воздействия. Количество элементов в сетке и временной интервал моделирования – параметры программы. Программа должна демонстрировать ускорение по сравнению с последовательным вариантом. Предусмотреть визуализацию результатов посредством утилиты gnuplot. При этом утилита gnuplot должна вызываться отдельной командой после окончания расчета.

Описание структуры программы

В начале программы происходит инициализация окружения с помощью *MPI_Init()*. Определяется количество процессов в данной коммуникационной области, а также идентификатор данного процесса в ней. Происходит считывание аргументов командной строки – размера сетки и модельного времени, выделяется требуемая память. После этого корневым процессом вызывается функция, устанавливающая граничные условия в сетке.

Для каждого временного слоя, требуемого для вычислений (для данной задачи это *t1* и *t*) созданы массивы. Двумерные динамические массивы не могут быть переданы процессам, поэтому были созданы одномерные массивы, причем

отдельно массив для текущего момента времени, и отдельно для следующего. Каждый процесс в определенные моменты времени хранит у себя указатели на определенную часть двух массивов (*z0* – значения *z* в предыдущий момент времени (*t - 1*), *z1* – в текущий момент (*t*)) и производит с их помощью вычисления.

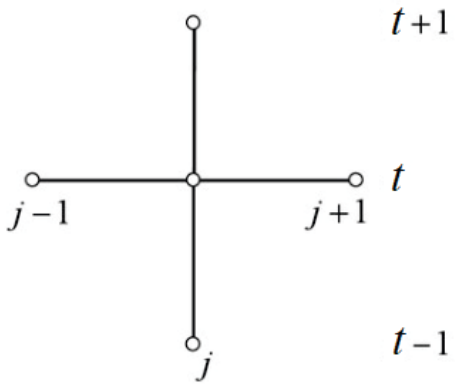

Рис. 1. Шаблон расчётной схемы

Корневой процесс хранит массивы *Z0* и *Z1* размерность которых равна *lengthX* – количеству узлов струны и извлекается из аргумента программы. Использование двух массивов обусловлено их различной логической функцией – один массив хранит значения *z* в момент времени *(t - 1)*, второй в момент времени *t*. Каждый процесс хранит малые массивы *z0* и *z1* имеющие размер *lengthX / total*, где *total* – общее количество процессов.

Перед началом итерации малые массивы *z0* и *z1* заполняются значениями из больших массивов *Z0* и *Z1*. В конце итерации происходит сбор значений с малых массивов в большие и запись в файл. При необходимости, запись в файл можно отключить, тогда для каждой итерации рассылка производиться не будет, а будет производиться обмен значениями, описанный ниже.

При вычислении одним из процессов крайнего узла своей части массива может возникнуть конфликт, так как ему нужно знать значения соседних узлов, один из которых принадлежит другому процессу. Этот конфликт разрешается следующим образом: каждый процесс перед каждым этапом вычислений средствами MPI отправляет значения своих крайних узлов соседним процессам.

Отправка значений узлов осуществляется с помощью функции передачи сообщений *MPI_Send*. Прием значений узлов осуществляется с помощью функция приема сообщений *MPI_Recv*. На каждом шаге по времени:

Все кроме 0-го процесса:

1. Отправляют предыдущему процессу значение *z* в крайнем левом узле своей расчетной ленты (расчетная лента – это массив узлов который был отправлен для расчета данному процессу).
2. Принимают сообщение от предыдущего процесса и заносят значение в переменную *zleft*.

После этих двух шагов в переменной *zleft* оказывается значение крайнего правого узла предыдущего процесса.

Все кроме последнего процесса:

1. Отправляют следующему процессу значение *z* в крайнем правом узле своей расчетной ленты.
2. Принимают сообщение от следующего процесса и заносят значение в переменную *zright*.

После этих двух шагов в переменной *zright* оказывается значение крайнего левого узла следующего процесса.

Функция вычислений реализована следующим образом. Одними из её параметров являются указатели на малые массивы *z0, z1* и указатели на переменные *zleft*, *zright*. С их помощью вычисляются значения функции *z* в новый момент времени (*t + 1*) и записываются в массив, хранящий значения в предыдущий момент времени (*t - 1*), т.к. он уже не актуален. На следующем временном шаге, передаваемые функции расчета в качестве параметров указатели на массивы следует поменять местами, чтобы значения *z* в момент времени *t* хранились в массиве для предыдущего момента, а значения в момент

времени ($t + 1$) хранились в массиве для текущего момента времени. Для этих целей в основном цикле используется счетчик чередования *znumber*.

Перед началом вычислений главный процесс рассылает всем остальным части массива узлов. Затем главный процесс запускает замер времени и все процессы приступают к расчету своей части. После окончания вычислений по всем временным шагам главный процесс останавливает замер времени и выводит на экран результат.

Если пользователь желает видеть результат моделирования на своем экране, то главный процесс после окончания расчета на каждом временном шаге собирает со всех процессов их части массива и выводит значения в файл. Затем пользователь может запустить скрипт утилиты gnuplot, сгенерированный в программе.

Программа завершается после вызова функции *MPI_Finalize()*, которая обеспечивает корректное завершение обменов в MPI.

Описание основных использованных структур

В корневом процессе хранятся массивы *Z0* и *Z1* размерностью *lengthX*, где *lengthX* равно общему количеству узлов струны.

Z0 = (double) calloc (lengthX, sizeof(double));*

Z1 = (double) calloc (lengthX, sizeof(double));*

В каждом процессе хранятся массивы *z0* и *z1* длиной *newlengthX*, где *newlengthX* равно количеству узлов струны на один процесс.

z0 = (double) calloc ((newlengthX, sizeof(double));*

zl = (double) calloc ((newlengthX, sizeof(double));*

Для передачи крайних узлов соседним процессам во всех процессах кроме 0-го используется переменная *zleft*. После приема сообщения с помощью *MPI_Recv* переменная хранит значение крайнего правого узла предыдущего

процесса. Во всех процессах кроме последнего используется переменная *zright*. После приема сообщения с помощью *MPI_Recv* переменная хранит значение крайнего левого узла следующего процесса.

Также в программе используются структуры *timeval* и *timezone* для замера времени выполнения расчетов.

Блок-схема

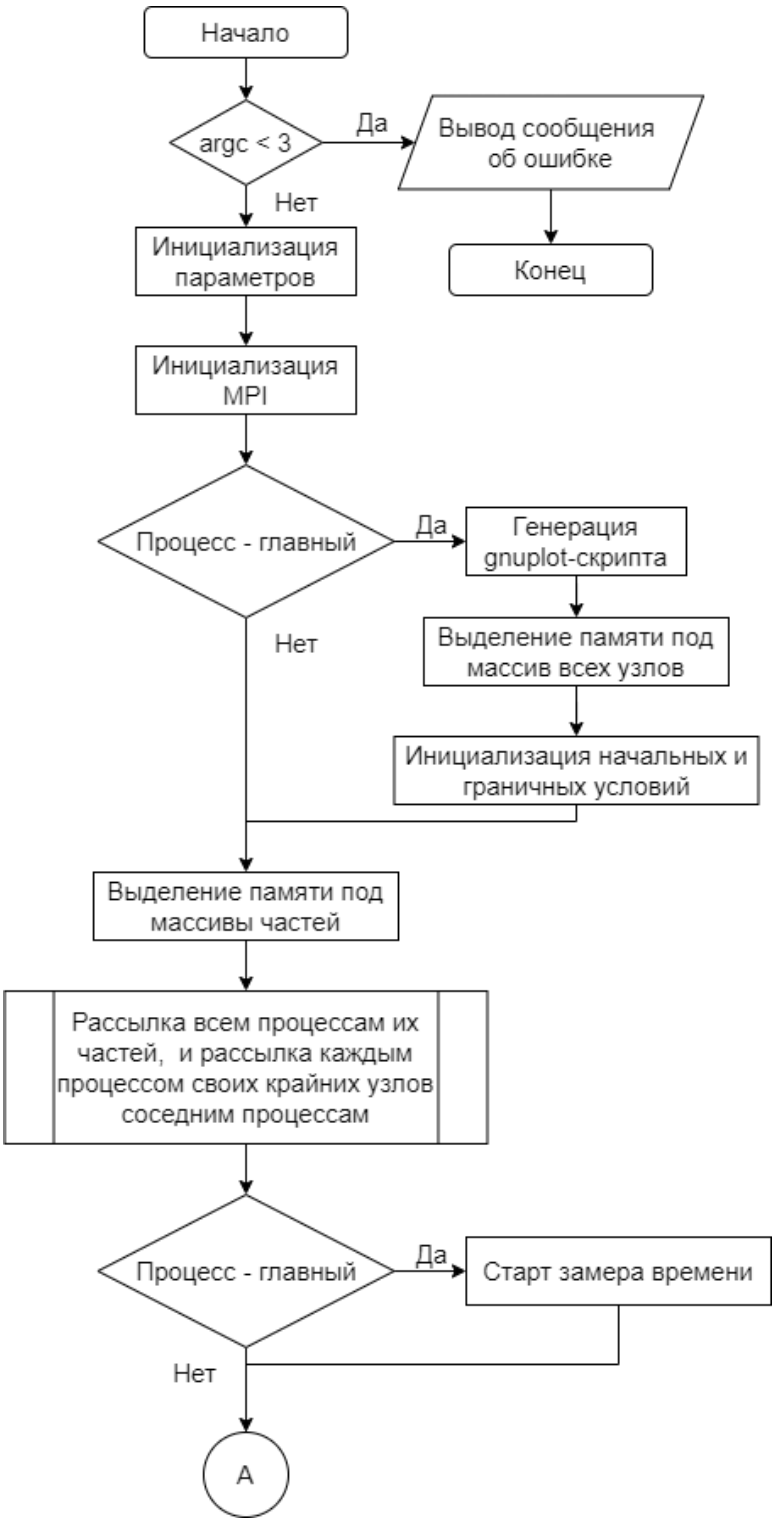

Рис. 2. Первая часть блок-схемы реализованного процесса

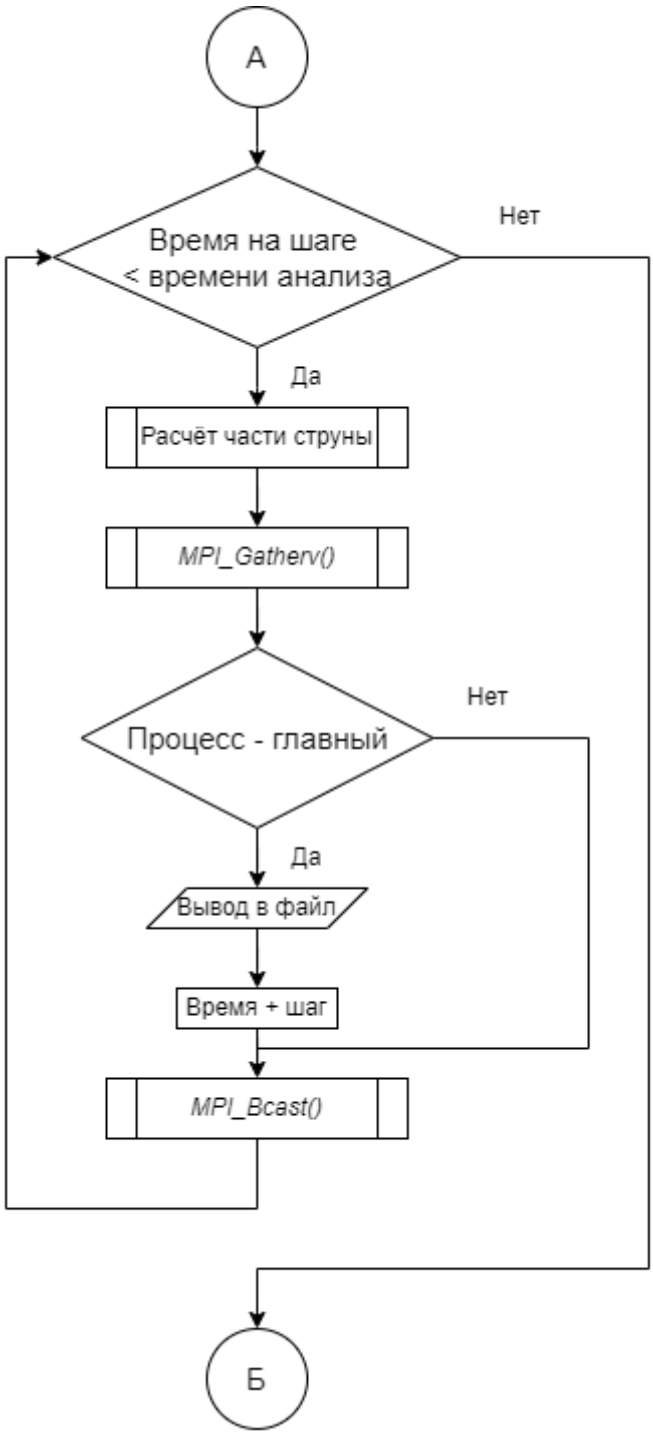

Рис. 3. Вторая часть блок-схемы реализованного процесса

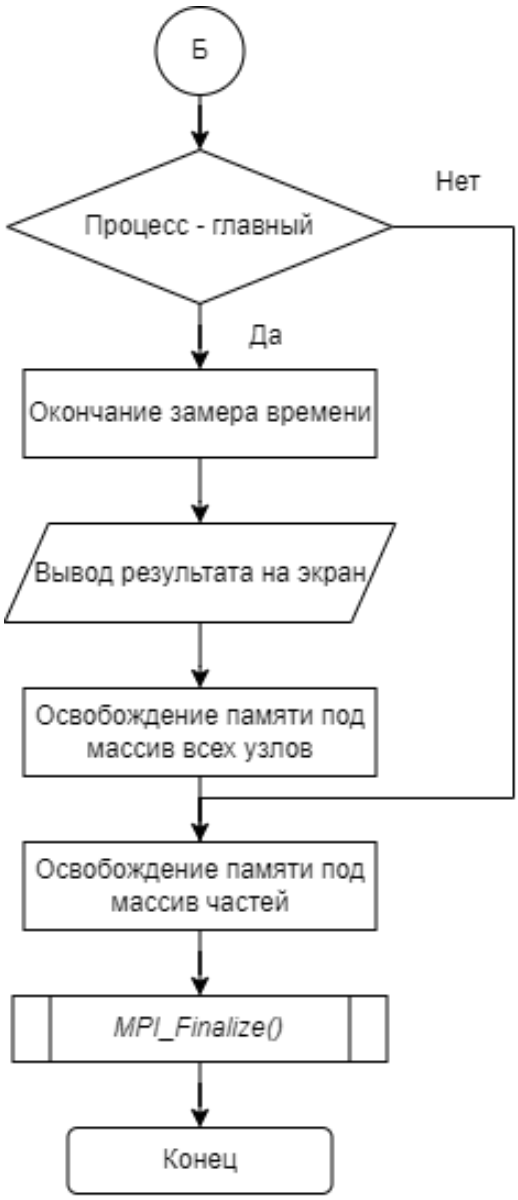

Рис. 4. Третья часть блок-схемы реализованного процесса

Пример работы программы

Запустим программу для струны с нулевыми начальными условиями и внешней силой, действующей вверх в первой четверти струны и вниз в третьей четверти. Ниже приведем один из кадров получившейся анимации.

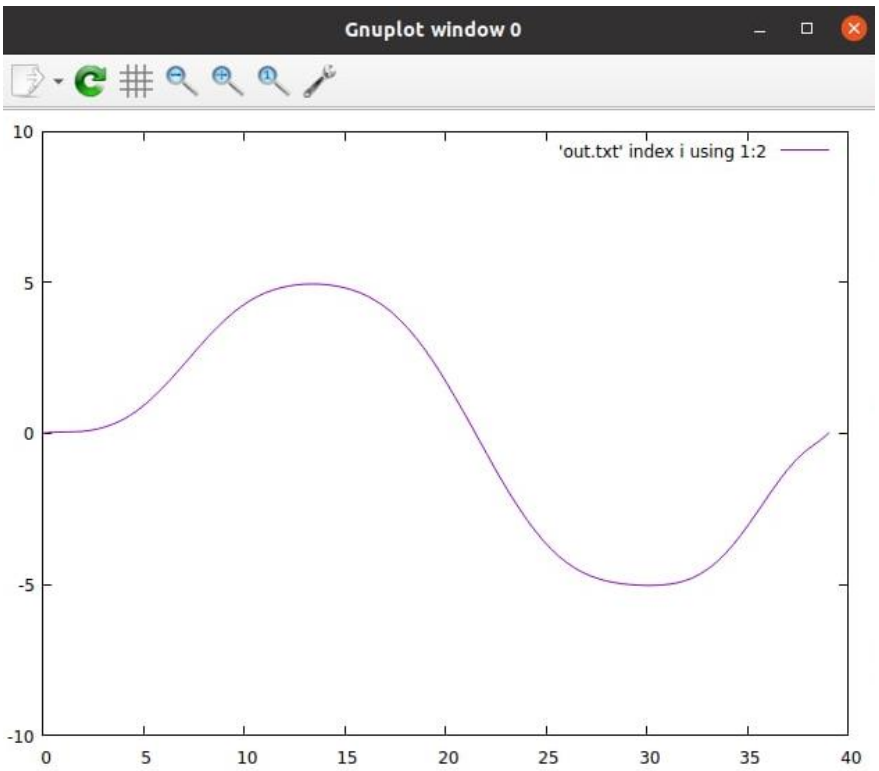

Рис. 5. Пример работы программы. Ось X – длина струны, ось Y – амплитуда колебания струны

Ускорение программы

В лабораторных условиях проведен эксперимент для проверки ускорения вычислений в зависимости от количества параллельно выполняющихся процессов. Параметры программы: 10000000 узлов, временной интервал 1000 сек.

Таблица 1 – Оценка временных затрат на работу программы при различном количестве процессов

Количество процессов	Время выполнения, с
1	111,101
2	53,067
4	28,54
8	16,325
16	17,141

По таблице 1 видно, что при увеличении количества потоков до 8, наблюдается существенный прирост производительности. Это обусловлено тем, что потоки успешно распараллеливаются, благодаря чему скорость вычислений растет.

При увеличении количества потоков до 16, производительность падает. Это связано с тем, что при распараллеливании тратятся дополнительные ресурсы на поддержку параллельной обработки, в частности, на переключение контекста процессора между потоками. Получается, дополнительные затраты появились, а потоки все равно не работают параллельно.

Текст программы

```
#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

#include <sys/time.h>

#include <math.h>

#include "mpi/mpi.h"
```

```
#define A 0.5

#define LEFT 1

#define RIGHT 2

#define LENGTHX 20

#define LENGTHT 200

#define LENGTHY 10

#define XRANGE 16

#define GNUPLOT


double dx;

double dt;

double dx2;

double dt2;

double a2;

FILE *out, *fp;


struct timeval tv1, tv2, dtv;

struct timezone tz;


void time_start()

{

    gettimeofday(&tv1, &tz);

}


void writeIntoFile(double *Z, int lengthX, double CurrentTime)

{

    int i, k;
```

```
    for (i = 0; i < lengthX; i++)
        fprintf(out, "%d\t%lf\n", i * (int)dx, Z[i]);
    fprintf(out, "\n");
    fprintf(out, "\n\n");
}

// Генерация скрипта Gnuplot
void GenerateGnuplot(int timeInterval)
{
    out = fopen("out.txt", "w");
    fp = fopen("script.dat", "w");
    fprintf(fp, "set cbrange [0.9:1]\n");
    fprintf(fp, "set xrange [0:%d]\n", (int)XRANGE);
    fprintf(fp, "set yrange [-%d:%d]\n", (int)LENGTHY, (int)LENGTHY);
    fprintf(fp, "set palette defined (1 '#ce4c7d')\n");
    fprintf(fp, "set style line 1 lc rgb '#b90046' lt 1 lw 0.5\n");
    fprintf(fp, "do for [i=0:%d] {\n", (int)LENGTHT - 1);
    fprintf(fp, "plot 'out.txt' index i using 1:2 smooth bezier\n");
    fprintf(fp, "pause 0.1}\npause -1\n");
}

// Задание граничных условий
void FirstCalculation(double *Z0, double *Z1, int lengthX)
{
    int i;

    for (i = 0; i < lengthX; i++)
    {
        Z0[i] = Z1[i] = 0;
```

```
    }
    writeIntoFile(Z0, lengthX, 0);
}

// Внешняя сила
double F(int x, double curtime, int a)
{
    if (curtime <= 5 && (int)x == a)
    {
        return 1;
    }
    return 0;
}

int time_stop()
{
    gettimeofday(&tv2, &tz);
    dtv.tv_sec = tv2.tv_sec - tv1.tv_sec;
    dtv.tv_usec = tv2.tv_usec - tv1.tv_usec;

    if (dtv.tv_usec < 0)
        dtv.tv_sec--;
    dtv.tv_usec += 1000000;

    return dtv.tv_sec * 1000 + dtv.tv_usec / 1000;
}

// Аргумент Z0 всегда текущий, а Z1 - предыдущий
// Z0 может быть как Z0, так и Z1
```

```
      void calculate(double *Z0, double *Z1, int lengthX, double CurrentTime, int myrank,
int total, double *zleft, double *zright)
      {
          int i;
          int index;

          if (myrank != 0)
          {
              index = 0;
              Z1[index] = dt2 * (a2 * (*zleft - 2 * Z0[index] + Z0[index + 1]) / dx2 +
                                  F(myrank * lengthX + index, CurrentTime, 10) -
                                  F(myrank * lengthX + index, CurrentTime, 25)) +
                          2 * Z0[index] - Z1[index];
              *zleft = Z1[index];
          }

          for (i = 1; i < lengthX - 1; i++)
          {
              index = i;
              Z1[index] = dt2 * (a2 * (Z0[index - 1] - 2 * Z0[index] + Z0[index + 1]) /
dx2 +
                                  F(myrank * lengthX + index, CurrentTime, 10) -
                                  F(myrank * lengthX + index, CurrentTime, 25)) +
                          2 * Z0[index] - Z1[index];
          }

          if (myrank != total - 1)
          {
              index = lengthX - 1;
```

```
            Z1[index] = dt2 * (a2 * (Z0[index - 1] - 2 * Z0[index] + *zright) / dx2 +
                               F(myrank * lengthX + index, CurrentTime, 10) -
                               F(myrank * lengthX + index, CurrentTime, 25)) +
                        2 * Z0[index] - Z1[index];
            *zright = Z1[index];
        }

        // Крайние узлы
        // Если это не последний процесс, то нужен правый ряд
        if (myrank != total - 1)
        {
            // Кидаем правую часть следующему процессу
            MPI_Send((void *)zright, 1, MPI_DOUBLE, myrank + 1, LEFT, MPI_COMM_WORLD);

            // Ловим левую часть следующего процесса
            // Для вычисляющего процесса это будет правой
            MPI_Recv((void *)zright, 1, MPI_DOUBLE, myrank + 1, RIGHT, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }

        // Если это не нулевой процесс, то нужно получить левый ряд
        if (myrank != 0)
        {
            // Кидаем левую часть предыдущему процессу
            MPI_Send((void *)zleft, 1, MPI_DOUBLE, myrank - 1, RIGHT, MPI_COMM_WORLD);

            // Ловим правую часть предыдущего процесса
            // Для вычисляющего процесса это будет левой
```

```
            MPI_Recv((void *)zleft, 1, MPI_DOUBLE, myrank - 1, LEFT, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }
    }

    int main(int argc, char **argv)
    {
        // Номер процесса, кол-во процессов
        int myrank, total;

        double *Z0;
        double *Z1;
        double *z0;
        double *z1;
        double zleft;
        double zright;
        double *tmp;

        if (argc < 3)
        {
            printf("./a.out <lengthX> <timeInterval>\n");
            exit(0);
        }

        // Сетка по x
        int lengthX = atoi(argv[1]);

        // Временной интервал
        double timeInterval = atof(argv[2]);
```

```
if (lengthX > LENGTHX)
    dx = 1.0;
else
    dx = LENGTHX / lengthX;

if (timeInterval > LENGTHT)
    dt = 1.0;
else
    dt = timeInterval / LENGTHT;

dx2 = dx * dx;
dt2 = dt * dt;
a2 = A * A;

// Текущий момент времени
double CurrentTime = dt;

// В какой массив записывать
int znumber = 1;

// Инициализация коммуникационных средств MPI
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &total);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

// Узлов на один процесс
int newlengthX = lengthX / total;
```

```
// Остаток
int lengthXmod = lengthX % total;

if (myrank == 0)
{
    GenerateGnuplot(timeInterval);

    // Значение узлов в текущий момент времени
    Z0 = (double *)calloc(lengthX, sizeof(double));

    // Значение узлов в предыдущий момент времени
    Z1 = (double *)calloc(lengthX, sizeof(double));

    FirstCalculation(Z0, Z1, lengthX);

    if (myrank == total - 1)
    {
        newlengthX += lengthXmod;
    }

    // Для каждого процесса (включая нулевой) выделяем память
    z0 = (double *)calloc(newlengthX, sizeof(double));
    z1 = (double *)calloc(newlengthX, sizeof(double));

    // Каждому процессу передадим необходимую часть
    int *sendArr;
    sendArr = (int *)calloc(total, sizeof(int));

    int i;
```

```
for (i = 0; i < total; i++)
    sendArr[i] = i * newlengthX;

int *sendArrCount;
sendArrCount = (int *)calloc(total, sizeof(int));

for (i = 0; i < total; i++)
    sendArrCount[i] = newlengthX;

sendArrCount[total - 1] += lengthXmod;

// Разбрызгиватель
MPI_Scatterv((void *)(Z0), sendArrCount, sendArr,
             MPI_DOUBLE, (void *)(z0), newlengthX,
             MPI_DOUBLE, 0, MPI_COMM_WORLD);

MPI_Scatterv((void *)(Z1), sendArrCount, sendArr,
             MPI_DOUBLE, (void *)(z1), newlengthX,
             MPI_DOUBLE, 0, MPI_COMM_WORLD);

zleft = z0[0];
zright = z0[newlengthX - 1];

// Если это не последний процесс, то нужен правый
if (myrank != total - 1)
{
    // Кидаем правую часть следующему процессу
```

```
                MPI_Send((void *)&zright, 1, MPI_DOUBLE, myrank + 1, LEFT,
MPI_COMM_WORLD);

                // Ловим левую часть следующего процесса
                MPI_Recv((void *)&zright, 1, MPI_DOUBLE, myrank + 1, RIGHT,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }

            // Если это не нулевой процесс, то нужно получить левый ряд
            if (myrank != 0)
            {
                // Кидаем левую часть предыдущему процессу
                MPI_Send((void *)&zleft, 1, MPI_DOUBLE, myrank - 1, RIGHT,
MPI_COMM_WORLD);

                // Ловим правую часть предыдущего процесса
                // Для вычисляющего процесса это будет левой
                MPI_Recv((void *)&zleft, 1, MPI_DOUBLE, myrank - 1, LEFT,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }

            if (myrank == 0)
                time_start();

            while (CurrentTime < timeInterval)
            {
                if (znumber == 1)
                    calculate(z0, z1, newlengthX, CurrentTime, myrank, total, &zleft,
&zright);
                else
```

```
                    calculate(z1, z0, newlengthX, CurrentTime, myrank, total, &zleft,
&zright);

        #ifdef GNUPLOT
                    if (znumber == 1)
                    {
                        MPI_Gatherv((void *)z1, newlengthX, MPI_DOUBLE,
                                    (void *)(Z1), sendArrCount, sendArr,
                                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
                        if (myrank == 0)
                            // Запись в gnuplot
                            writeIntoFile(Z1, lengthX, CurrentTime);
                    }
                    else
                    {
                        MPI_Gatherv((void *)z0, newlengthX, MPI_DOUBLE,
                                    (void *)(Z0), sendArrCount, sendArr,
                                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
                        if (myrank == 0)
                            // Запись в gnuplot
                            writeIntoFile(Z0, lengthX, CurrentTime);
                    }

                    // Ждем запись
                    MPI_Barrier(MPI_COMM_WORLD);
        #endif
                    if (!myrank)
                    {
                        znumber *= -1;
```

```
                CurrentTime += dt;
            }

            MPI_Bcast((void *)&znumber, 1, MPI_INT, 0, MPI_COMM_WORLD);

            // Делимся со всеми процессами значением времени на текущем
            // шаге, это нужно для того, чтобы правильно рассчитывалось
            // внешнее воздействие на каждый узел, которое зависит от
            // координаты и времени
            MPI_Bcast((void *)&CurrentTime, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }

        // Вывод времени работы
        if (myrank == 0)
        {
            int ms = time_stop();
            printf("Time: %d milliseconds\n", ms);
            free(Z0);
            free(Z1);
        }
        // Освобождение выделенной памяти
        free(z0);
        free(z1);
        free(sendArr);
        free(sendArrCount);
        MPI_Finalize();
        exit(0);
    }
}
```