Филиал МГУ имени М.В. Ломоносова в городе Сарове

Направление подготовки
«Фундаментальная информатика и информационные технологии»

Студеникина Мария Александровна

Параллельная программа для уравнения колебания струны (MPI)

ОТЧЕТ

Саров, 2023

Содержание

Постановка задачи

Написать параллельную программу (с использованием технологии MPI) для численного решения двумерного нестационарного неоднородного уравнения колебания струны в прямоугольной области.

Следует использовать явную разностную схему.

Описание точного аналитического решения

$$\frac{\partial^2 u}{\partial t^2} = a^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + f(x, y, t)$$

$$u(t, x, y) = cos(2 * t) * sin(3 * x) * cos(4 * y)$$

Описание метода решения

Уравнение колебаний струны в прямоугольной области имеет такой вид:

$$\frac{\partial^2 u}{\partial t^2} = a^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + f$$

$$u = u(t, x, t)$$

$$f = f(x, y, t)$$

Граничные условия можно задать таким образом:

$$u(t, 0, y) = \mu_1(t)$$

$$u(t, x, 0) = \mu_2(t)$$

$$u(t, L_1, y) = \mu_3(t)$$

$$u(t, x, L_2) = \mu_4(t)$$

Первый слой:

$$\frac{u_{i,j}^1 - u_{i,j}^0}{\tau} = u_{i,j}^0 + \frac{\tau}{2}\left(a^2\left(\frac{\partial^2 u_{i,j}^0}{\partial x^2} + \frac{\partial^2 u_{i,j}^0}{\partial y^2}\right) + f_{i,j}^0\right.$$

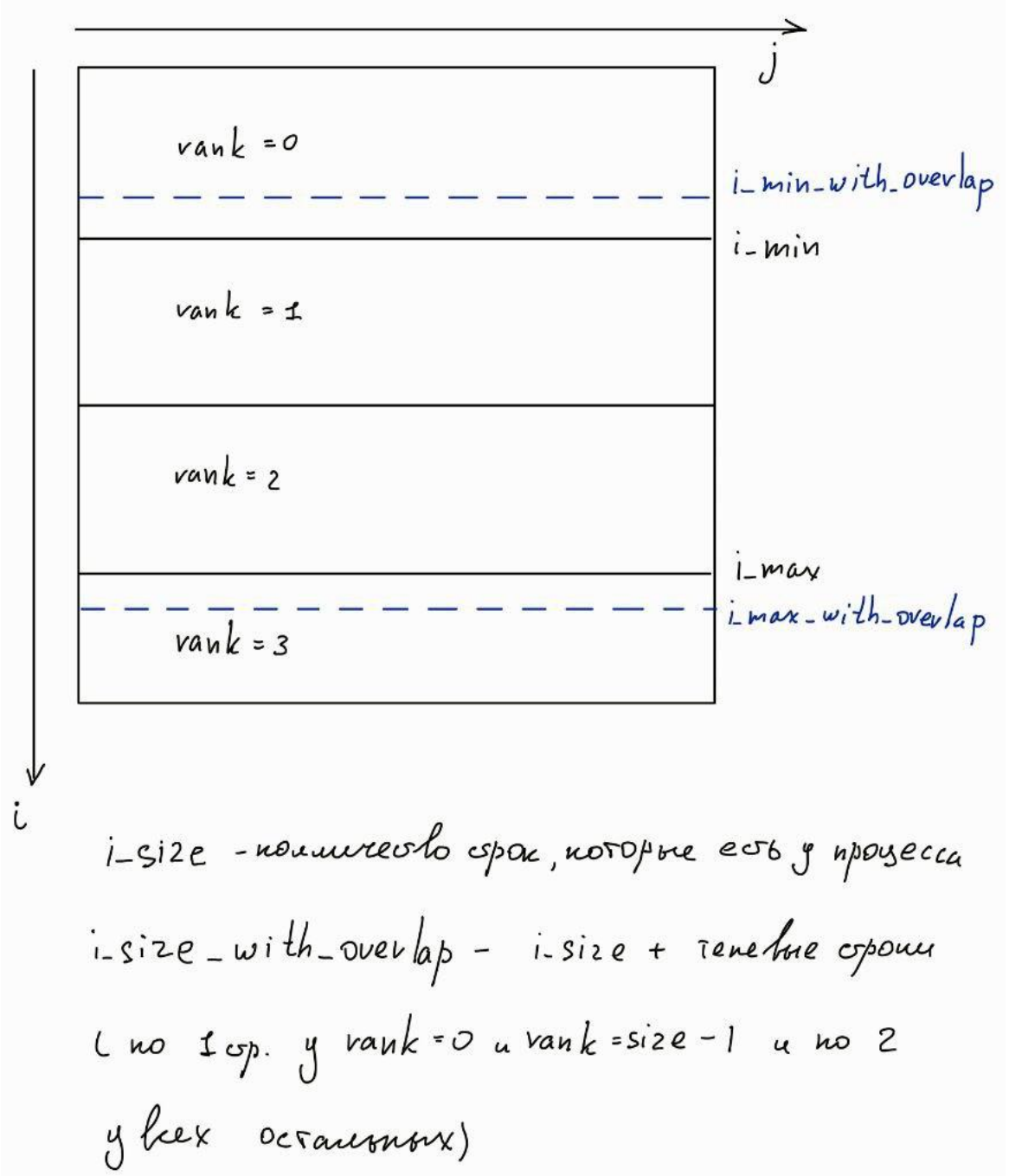
$$u_{i,j}^1 = u_{i,j}^0 + \tau\left(u_{i,j}^0 + \frac{\tau}{2}\left(a^2\left(\frac{\partial^2 u_{i,j}^0}{\partial x^2} + \frac{\partial^2 u_{i,j}^0}{\partial y^2}\right) + f_{i,j}^0\right)\right.$$

Следующие слои:

$$\frac{u_{i,j}^{n+1} - 2u_{i,j}^n + u_{i,j}^{n-1}}{\tau^2} = a^2\left(\frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{h_x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{h_y^2}\right) + f_{i,j}^n$$

$$u_{i,j}^{n+1} = 2u_{i,j}^n - u_{i,j}^{n-1} + \tau^2\left(a^2\left(\frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{h_x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{h_y^2}\right) + f_{i,j}^n\right)$$

Декомпозиция области



Описание используемой вычислительной системы

Число узлов	7 узлов	7
Число процессов	2 процесса на узле	14
Число ядер	каждый процессор 18-ядерный	252

Аналитическая зависимость ожидаемого времени решения от параметров задачи и от параметров вычислительной системы

$$(N * K * t * Mx * My)/p + 4 * Mx * ts * N$$

Ошибки и погрешности

$\tau \backslash h$	0.002	0.001	0.0005	0.00025
0.001	1.312e-06	-	-	-
0.0005	1.312e-06	3.279e-07	-	-
0.00025	1.309e-06	3.272e-07	8.179e-08	-
0.000125	1.297e-06	3.242e-07	8.106e-08	2.027e-08

Таблица 1. Значение нормы (norm_L_2)
в зависимости от шага по времени и по пространству

$\tau \backslash h$	0.002	0.001	0.0005	0.00025
0.001	2.827e-06	-	-	-
0.0005	2.825e-06	7.063e-07	-	-
0.00025	2.819e-06	7.047e-07	1.762e-07	-
0.000125	2.794e-06	6.984e-07	1.746e-07	4.365e-08

Таблица 2. Значение нормы (norm_c)
в зависимости от шага по времени и по пространству

Эффективность распараллеливания

размер сетки по t	размер сетки по x и y	кол-во процессов	время OpenMP	время MPI	аналитическое время
4000	2000	1	1000.90387	169.51779	208.69562
		2	549.84874	88.24003	104.54651
		4	288.70864	45.46285	52.37259
		8	148.47438	23.89962	26.28564
		16	75.64959	13.89627	13.24216
		32	41.55928	7.88880	6.72042
		36	38.94555	7.63674	5.99578
2000	1000	1	121.69534	21.26199	26.08696
		2	68.12887	11.32497	13.09314
		4	36.48885	6.01669	6.57141
		8	18.70931	3.17903	3.3105
		16	9.82428	2.00716	1.68011
		32	5.49401	1.55153	0.86489
		36	5.15663	1.17637	0.77431

Таблица 3. Время выполнения программы с использованием OpenMP и MPI в зависимости от размера сетки по времени и по пространству

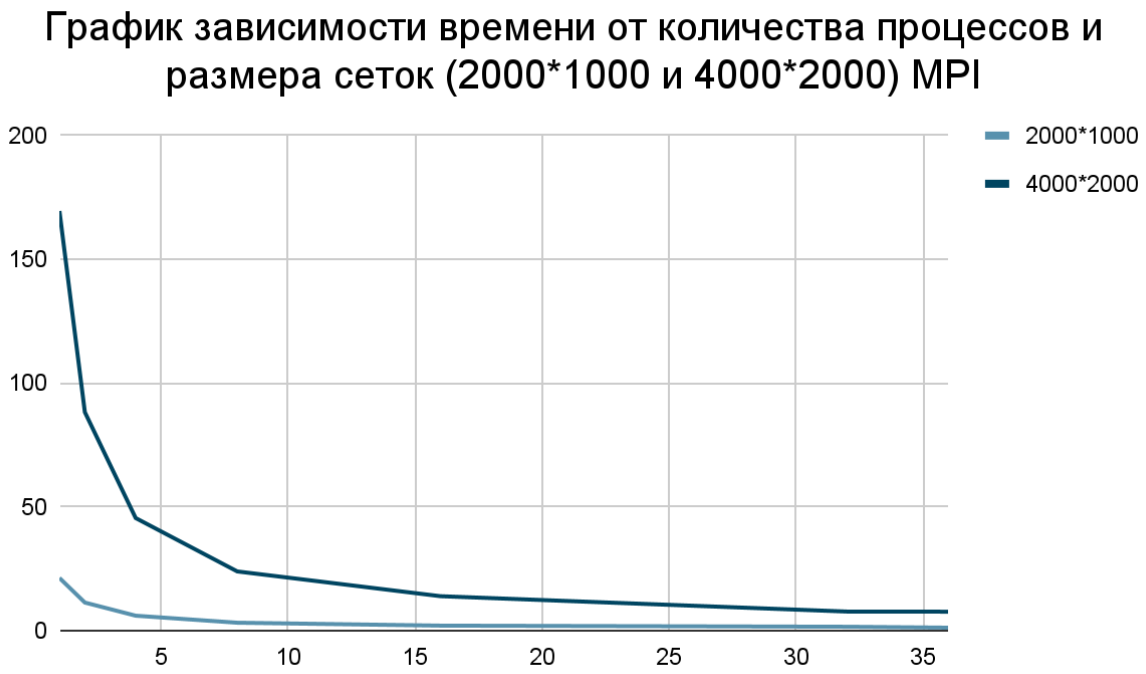
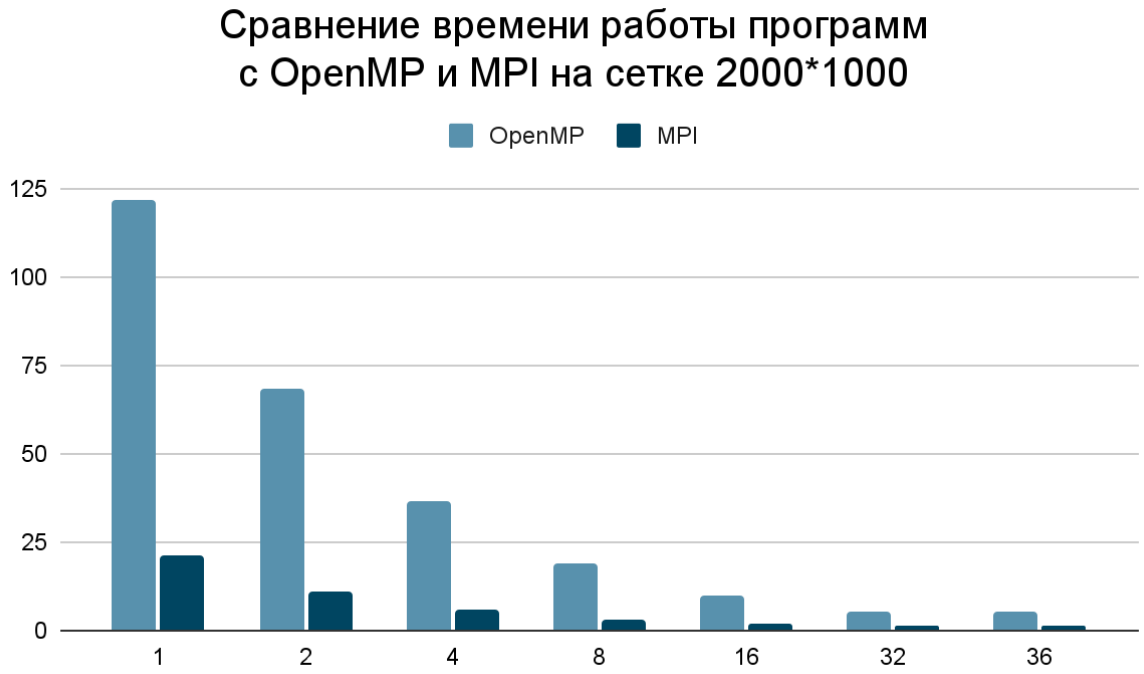
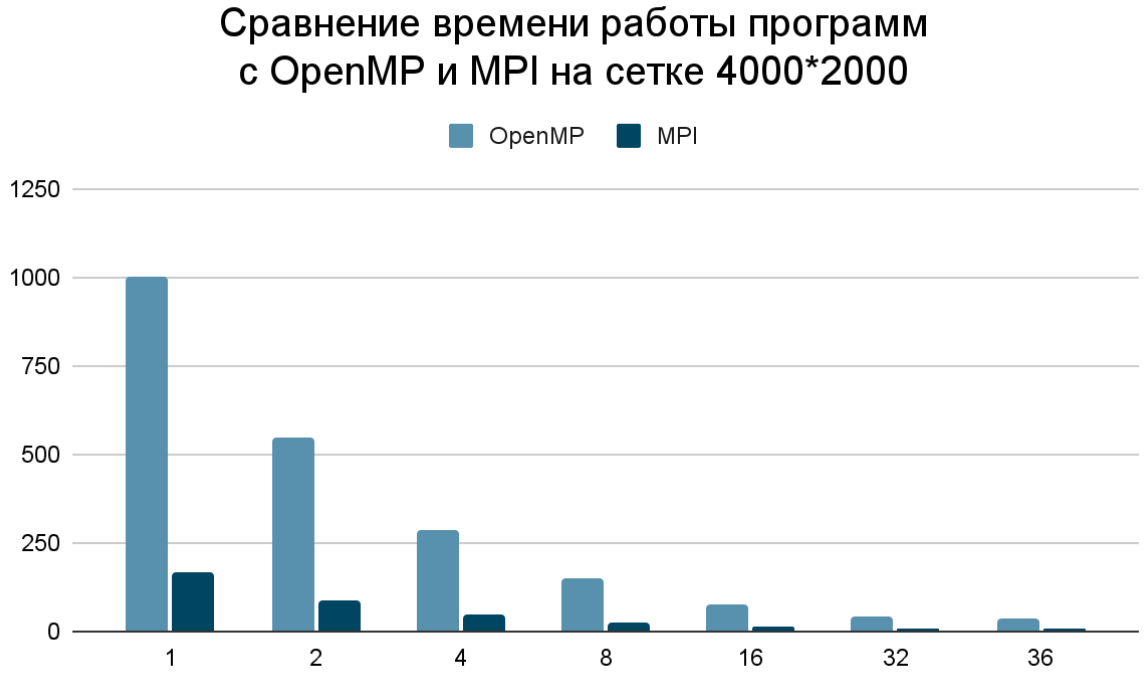


График зависимости времени от количества процессов и размера сеток (2000*1000 и 4000*2000) MPI
200
150
100
50
0
5
10
15
20
25
30
35
2000*1000
4000*2000


График зависимости времени от количества процессов и размера сеток (2000*1000 и 4000*2000) OpenMP
1250
1000
750
500
250
0
5
10
15
20
25
30
35
2000*1000
4000*2000


Сравнение времени работы программ
с OpenMP и MPI на сетке 2000*1000
OpenMP
MPI
125
100
75
50
25
0
1
2
4
8
16
32
36


Сравнение времени работы программ
с OpenMP и MPI на сетке 4000*2000
OpenMP
MPI
1250
1000
750
500
250
0
1
2
4
8
16
32
36

Компиляция

MPI
mpicc str_mpi.c -lm
sbatch example_MPI

OpenMP
gcc str_omp.c -fopenmp -lm
sbatch example_OMP

Заключение

Распараллеливание программы с помощью MPI дало больший выигрыш по времени в сравнении с OpenMPI.

Можно заметить, что с уменьшением значений t и h, ошибки также уменьшаются. Это можно объяснить более точным приближением функции к истинному значению при меньшем шаге пространства и времени.

Также можно заметить, что значения ошибок примерно одинаковы для близких значений t и h. Таким образом, можно сделать вывод о том, что более мелкий шаг пространства и времени приводит к более точному приближению значения.

Приложение

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define TAG_TOP_TO_BOT 123
#define TAG_BOT_TO_TOP 321

double phi(double x, double y)
{
   return sin(3 * x) * cos(4 * y);
}

double psi_mu_1(double x, double y)
{
   return 0;
}

double f(double a, double t, double x, double y)
{
   return (25 * a * a - 4) * cos(2 * t) * sin(3 * x) * cos(4 * y);
}

double mu_2(double t, double x)
{
   return cos(2 * t) * sin(3 * x);
}

double mu_4(double t, double x, double Ly)
{
   return cos(2 * t) * sin(3 * x) * cos(4 * Ly);
}

int main(int argc, char** argv)
{
   MPI_Init(&argc, &argv);

   int size, rank;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   int N, Mx, My;
   N = atoi(argv[1]),
```

```
    Mx = atoi(argv[2]),
    My = atoi(argv[3]);

    double T, Lx, Ly;
    T = 1; // сетка по времени от [0,T]
    Lx = 1;
    Ly = 1; // сетка по пространству x и y [0,L]

    double tau, hx, hy;
    tau = T / N; // шаг по времени
    hx = Lx / Mx; // шаг по пространству x
    hy = Ly / My; // шаг по пространству y

    if(tau < sqrt(2) * hx * hx)
    {
        if(rank == 0)
        {
            printf("ERROR: tau < sqrt(2) * hx\n");
        }
        MPI_Finalize();
        return 1;
    }

    double a = 1;

    double norm_L_2 = 0, norm_c = 0; // нормы
    double norm_L_2_global, norm_c_global;

    const int
        i_size = (My+1) / size + (rank < (My+1) % size ? 1 : 0),
// количество строк, которые есть у процесса
        i_size_with_overlap = i_size + (!rank || rank == size-1 ? 1 : 2),
// кол-во строк + теневые строки (по 1 у 0 и последнего и по 2 у всех остальных)
        i_min = rank < (My+1) % size ? i_size * rank : (My+1) - (size-rank)*i_size,
// первая строка, которая есть у процесса
        i_max = i_min + i_size,
// последняя строка, которая есть у процесса
        i_min_with_overlap = i_min - (rank > 0),
// теневая строка до
        i_max_with_overlap = i_max + (rank < size-1);
// теневая строка после

    double** u_0;
    double** u_1;
    double** u_2;
```

```
   // константы для упрощения подсчета формул
   double con_1, con_2, con_3, con_4;
   con_1 = tau * tau / 2 * a * a;
   con_2 = con_1 / (hx * hx);
   con_3 = con_1 * 2;
   con_4 = con_2 * 2;

   u_0 = malloc((i_size_with_overlap + 1) * sizeof(double*));
   u_1 = malloc((i_size_with_overlap + 1) * sizeof(double*));
   u_2 = malloc((i_size_with_overlap + 1) * sizeof(double*));

   for (int i = 0; i <= i_size_with_overlap; ++i)
   {
       u_0[i] = malloc((Mx + 1) * sizeof(double));
       u_1[i] = malloc((Mx + 1) * sizeof(double));
       u_2[i] = malloc((Mx + 1) * sizeof(double));
   }

   double t = 0; // текущее время
   double time1, time2;
   MPI_Barrier(MPI_COMM_WORLD);
   time1 = MPI_Wtime();

   for(int i_2 = i_min_with_overlap; i_2 < i_max_with_overlap; i_2++)
   {
       for(int j = 0; j < Mx + 1; j++)
       {
           const int i = i_2 - i_min_with_overlap;
           u_1[i][j] = phi(j * hx, i_2 * hy);
       }
   }

   for(int i_2 = fmax(1, i_min); i_2 < fmin(My, i_max); i_2++) // все кроме
граничных условий
   {
       for(int j = 1; j < Mx; j++)
       {
           const int i = i_2 - i_min_with_overlap;
           u_2[i][j] = u_1[i][j] + tau * psi_mu_1(j * hx, i_2 * hy)
           + con_2 * (u_1[i + 1][j] - 2 * u_1[i][j] + u_1[i - 1][j]
           + u_1[i][j + 1] - 2 * u_1[i][j] + u_1[i][j - 1]) + con_1 * f(a, t, j *
hx, i_2 * hy);
       }
   }
```

```
t += tau;

// граничные условия (для распаралеленого случая)
// самый левый столбец и самый правый столбец
if(size != 1)
{
    for(int i_2 = fmax(1, i_min); i_2 < fmin(My, i_max); i_2++)
    {
        const int i = i_2 - i_min_with_overlap;
        u_2[i][0] = psi_mu_1(t, Lx);
        u_2[i][Mx] = mu_4(t, Lx, i_2 * hy);
    }

    // самая верхняя строка и самая нижняя строка
    if (rank == 0)
    {
        for (int j = 0; j <= Mx; ++j)
        {
            u_2[i_min - i_min_with_overlap][j] = mu_2(t, j * hx);
        }
    }
    else if (rank == size - 1)
    for(int j = 0; j < Mx + 1; j++)
    {
        u_2[i_max - 1 - i_min_with_overlap][j] = mu_4(t, j * hx, Ly);
    }
}

// граничные условия (для нераспаралеленого случая)
if(size == 1)
{
    // самый левый столбец и самый правый столбец
    for(int i = 0; i < My + 1; i++)
    {
        u_2[i][0] = psi_mu_1(t, Lx);
        u_2[i][Mx] = mu_4(t, Lx, i * hy);
    }
    // самая верхняя строка и самая нижняя строка
    for(int j = 0; j < Mx + 1; j++)
    {
        u_2[0][j] = mu_2(t, j * hx);
        u_2[My][j] = mu_4(t, j * hx, Ly);
    }
}
```

```
   // ищем соседей для процесса
   const int bot = (rank - 1 < 0) ? (MPI_PROC_NULL) : (rank - 1);          // сосед
rank - 1
   const int top = (rank + 1 >= size) ? (MPI_PROC_NULL) : (rank + 1);      // сосед
rank + 1

   while(t < T - tau / 2)
   {
       // асинхронная передача и прием сообщения (пересылки крайних данных, те
которые находятся на границах)
       // top и bot - идентификатор получателей и отправителей
       // TAG_TOP_TO_BOT и TAG_BOT_TO_TOP - тег сообщения, чтобы понимать кому
сообщение
       MPI_Request request[4];
       MPI_Irecv(u_2[i_max - i_min_with_overlap], Mx+1, MPI_DOUBLE, top,
TAG_TOP_TO_BOT, MPI_COMM_WORLD, &request[0]);
       MPI_Irecv(u_2[i_min_with_overlap - i_min_with_overlap], Mx+1, MPI_DOUBLE,
bot, TAG_BOT_TO_TOP, MPI_COMM_WORLD, &request[1]);
       MPI_Isend(u_2[i_max-1 - i_min_with_overlap], Mx+1, MPI_DOUBLE, top,
TAG_BOT_TO_TOP, MPI_COMM_WORLD, &request[2]);
       MPI_Isend(u_2[i_min - i_min_with_overlap], Mx+1, MPI_DOUBLE, bot,
TAG_TOP_TO_BOT, MPI_COMM_WORLD, &request[3]);
       MPI_Waitall(4, request, MPI_STATUS_IGNORE);

       double** tmp = u_0;
       u_0 = u_1;
       u_1 = u_2;
       u_2 = tmp;

       for(int i_2 = fmax(1, i_min); i_2 < fmin(My, i_max); i_2++)
       {
           for(int j = 1; j < Mx; j++)
           {
               const int i = i_2 - i_min_with_overlap;
               u_2[i][j] = 2 * u_1[i][j] - u_0[i][j]
               + con_4 * (u_1[i + 1][j] - 2 * u_1[i][j] + u_1[i - 1][j]
               + u_1[i][j + 1] - 2 * u_1[i][j] + u_1[i][j - 1]) + con_3 * f(a, t, j
* hx, i_2 * hy);
           }
       }
       t += tau;

       // граничные условия (для распаралеленого случая)
       // самый левый столбец и самый правый столбец
       if(size != 1)
```

```
    {
        for(int i_2 = fmax(1, i_min); i_2 < fmin(My, i_max); i_2++)
        {
            const int i = i_2 - i_min_with_overlap;
            u_2[i][0] = psi_mu_1(t, Lx);
            u_2[i][Mx] = mu_4(t, Lx, i_2 * hy);
        }

        // самая верхняя строка и самая нижняя строка
        if (rank == 0)
        {
            for(int j = 0; j < Mx + 1; j++)
            {
                u_2[i_min - i_min_with_overlap][j] = mu_2(t, j * hx);
            }
        }
        else if (rank == size - 1)
        {
            for(int j = 0; j < Mx + 1; j++)
            {
                u_2[i_max-1 - i_min_with_overlap][j] = mu_4(t, j * hx, Ly);
            }
        }
    }

    // граничные условия (для нераспаралеленого случая)
    if(size == 1)
    {
        // самый левый столбец и самый правый столбец
        for(int i = 0; i < My + 1; i++)
        {
            u_2[i][0] = psi_mu_1(t, Lx);
            u_2[i][Mx] = mu_4(t, Lx, i * hy);
        }
        // самая верхняя строка и самая нижняя строка
        for(int j = 0; j < Mx + 1; j++)
        {
            u_2[0][j] = mu_2(t, j * hx);
            u_2[My][j] = mu_4(t, j * hx, Ly);
        }
    }
}
MPI_Barrier(MPI_COMM_WORLD);
time2 = MPI_Wtime();
```

```
    // считаем нормы для того, чтобы сравнить с аналитическим решение
    for(int i_2 = i_min; i_2 < i_max; i_2++)
    {
        for(int j = 0; j <= Mx; ++j)
        {
            const int i = i_2 - i_min_with_overlap;
            const double u_t = mu_4(T, j * hx, i_2 * hy) -  u_2[i][j];
            norm_L_2 += u_t * u_t;
            norm_c = fmax(norm_c, fabs(u_t));
        }
    }

    // суммируем все значения norm_L_2 и сохраняем в norm_L_2_global
    MPI_Reduce(&norm_L_2, &norm_L_2_global, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (rank == 0)
    {
        norm_L_2_global = sqrt(hx * hy * norm_L_2_global);
    }

    // ищем максимальное значение norm_c и сохраняем в norm_c_global
    MPI_Reduce(&norm_c, &norm_c_global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    for (int i = 0; i < i_size_with_overlap; ++i)
    {
        free(u_0[i]);
        free(u_1[i]);
        free(u_2[i]);
    }
    free(u_0);
    free(u_1);
    free(u_2);

    if (rank == 0)
    {
        printf("%.3le, %.3le, Time = %.5lf\n", norm_L_2_global, norm_c_global, time2
- time1);
    }
    MPI_Finalize();
    return 0;
}
```