Федеральное государственное бюджетное образовательное учреждение высшего образования
"Уфимский государственный авиационный технический университет"

Кафедра Высокопроизводительных вычислительных технологий и систем

Дисциплина: Параллельное программирование

Отчет по лабораторной работе № 3

Тема: «Параллельный алгоритм решения краевой задачи для уравнения теплопроводности»

Группа ПМ-353	Фамилия И.О.	Подпись	Дата	Оценка
Студент	Шамаев И.Р.			
Принял	Юлдашев А.В.			

Уфа 2022

Цель: научиться использовать принцип геометрического параллелизма для разработки и программной реализации параллельных алгоритмов решения краевых задач для уравнений эволюционного типа на примере параллельной реализации явной численной схемы для решения краевой задачи уравнения теплопроводности.

Теоретический материал.

MPI (Message Passing Interface) – это стандартизованная библиотека функций, призванная обеспечить совместную работу параллельных процессов путем организации передачи сообщений между ними.

Интерфейс MPI:

- Модель программирования MPI.
- Базовые функции MPI.
- Функции парного взаимодействия.
- Функции коллективного взаимодействия.
- Функция определения времени
- Пользовательские типы данных.
- Управление областью взаимодействия и группой процессов.

Базовые понятия MPI

- MPI предназначен для написания программ для MIMD архитектур
- Каждая программа представляет собой совокупность одновременно работающих процессов, которые могут обмениваться сообщениями.
- Все процессы объединяются в группы.
- Обмен сообщениями возможен между процессами одной группы, которой поставлена в соответствие своя область связи.
- Идентификатор области связи называется коммуникатором.
- При запуске программы все доступные ей процессы объединяются в начальную группу с общей областью связи, имеющей коммуникатор MPI_COMM_WORLD.

Базовые операции стандарта MPI

- Операции межпроцессорного взаимодействия типа
- «точка-точка».
- Операции коллективного взаимодействия.
- Операции над группами процессов.
- Операции с областями коммуникации.
- Операции с топологией процессов.
- Функции ввода/вывода (появились в MPI-2.0)

Особенности реализации MPI для C/C++

1. Первая строка программы

#include “mpi.h”

2. В MPI принят ANSI C стандарт.
3. Нумерация массивов начинается с 0.
4. Массивы хранятся по строкам.

5. Логические переменные являются переменными типа integer со значением 0 в случае false и любым не нулевым значением, обозначающем true.

Базовые функции MPI

1. int ***MPI_Init***(int *argc, char **argv[]); – инициализация параллельной части программы.

Возвращает предопределенные константы

MPI_SUCCESS - возвращается в случае успешного выполнения,

MPI_ERR_ARG - ошибка неправильного задания аргумента,

MPI_ERR_INTERN - внутренняя ошибка (нехватка памяти),

MPI_ERR_UNKNOWN - неизвестная ошибка.

2. int ***MPI_Finalize***(void); завершение параллельной части программы.

3. int ***MPI_Comm_size***(MPI_Comm comm, int* size); определение числа процессов size в коммуникационной группе с коммуникатором comm.

4. int ***MPI_Comm_rank***(MPI_comm comm, int* rank); определение номера rank вызвавшего ее процесса, входящего в коммуникационную группу с коммуникатором comm

5. int ***MPI_Send***(void* sbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) – передача сообщений от одного процесса к другому

sbuf – адрес в памяти, начиная с которого размещаются передаваемые данные;

count – количество передаваемых элементов;

datatype – тип передаваемых элементов;

dest – номер процесса-получателя сообщения;

tag – метка передаваемого сообщения;

comm – коммуникатор.

6. int ***MPI_Recv***(void* rbuf, int count, MPI_Datatype datatype, int source, int tag, MPI_comm comm, MPI_Status *status) – прием сообщений от одного процесса к другому

Входные параметры:

count – количество получаемых элементов;

datatype – тип получаемых элементов;

source – номер процесса-отправителя сообщения;

tag – метка принимаемого сообщения;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти, начиная с которого размещаются принимаемые данные;

status – структура, содержащая информацию о принятом сообщении.

Структура status имеет три поля.

Status.MPI_SOURCE - номер процесса-отправителя;

Status.MPI_TAG - метка принимаемого сообщения;

Status.MPI_ERROR - код завершения приема сообщения.

Функция совмещенного приема/передачи.

int ***MPI_Sendrecv***(void* sbuf, int scount, MPI_Datatype sdatatype, int dest, int stag, void* rbuf, int rcount, MPI_Datatype rdatatype, int source, int rtag, MPI_comm comm, MPI_Status *status);

Входные параметры:

sbuf – адрес в памяти, начиная с которого размещаются передаваемые данные;

scount – количество передаваемых элементов;

sdatatype – тип отправляемых элементов;

dest – номер процесса-получателя сообщения;

stag – метка отправляемого сообщения;

rcount – количество получаемых элементов;

rdatatype – тип получаемых элементов;

source – номер процесса-отправителя сообщения;

rtag – метка принимаемого сообщения;

comm – коммуникатор.

Выходные параметры:

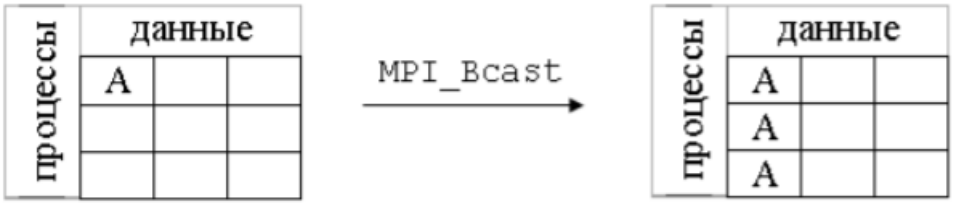
rbuf – адрес в памяти, начиная с которого размещаются принимаемые данные;

status – структура, содержащая информацию о принятом сообщении.

Функции коллективного взаимодействия

1. int ***MPI_Bcast***(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

2. Функция предназначена для рассылки данных, хранящихся на одном процессе, всем остальным процессам группы.



Входные параметры:

buf – адрес в памяти, начиная с которого размещаются передаваемые данные;

count – количество рассылаемых элементов;

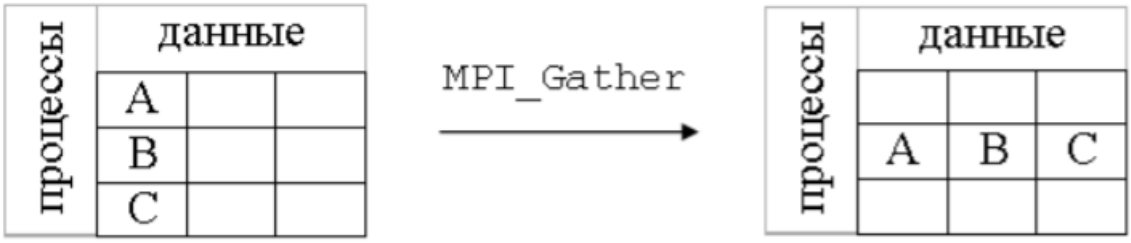
datatype – тип отправляемых данных;

root – номер процесса-отправителя сообщения;

comm – коммуникатор.

3. int ***MPI_Gather***(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm);

Функция предназначена для сбора данных со всех процессов на одном (так называемый, "совок“).



Входные параметры:

sbuf – адрес в памяти на каждом процессе, начиная с которого размещаются отправляемые данные;

scount – количество элементов, отправляемых с каждого процесса;

stype – тип отправляемых данных;

rcount – количество принимаемых элементов от каждого процесса;

rtype – тип принимаемых данных;

root – номер процесса, на котором осуществляется сборка сообщений;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти на процессе с номером root, начиная с которого размещаются принимаемые сообщения.

4. Вариант функции с варьируемым кол-вом собираемых элементов:

int ***MPI_Gatherv***(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int *rcounts, int *displs, MPI_Datatype rtype, int root, MPI_Comm comm);

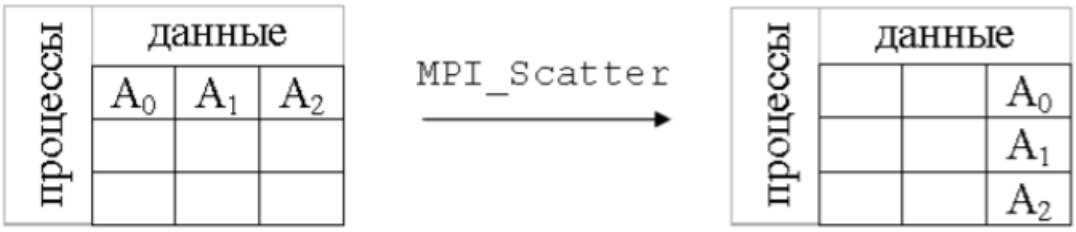
Характерные отличия:

rcounts – массив длин принимаемых от процессов сообщений;

displs – массив позиций в приемном буфере, по которым размещаются принимаемые сообщения.

5. int ***MPI_Scatter***(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm);

Функция предназначена для рассылки данных с одного процесса всем остальным процессам (так называемый, “разбрызгиватель”).



Входные параметры:

sbuf – адрес в памяти на процессе-отправителе сообщения, начиная с которого размещаются отправляемые данные;

scount – количество элементов, отправляемых каждому процессу;

stype – тип отправляемых данных;

rcount – количество элементов, принимаемых каждым процессом (длина принимаемого сообщения);

rtype – тип принимаемых данных;

root – номер процесса-отправителя сообщения;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти на каждом процессе, начиная с которого размещаются принимаемые сообщения.

6. Вариант функции с варьируемым кол-вом рассылаемых элементов:

int ***MPI_Scatterv***(void *sbuf, int *scounts, int *displs, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm);

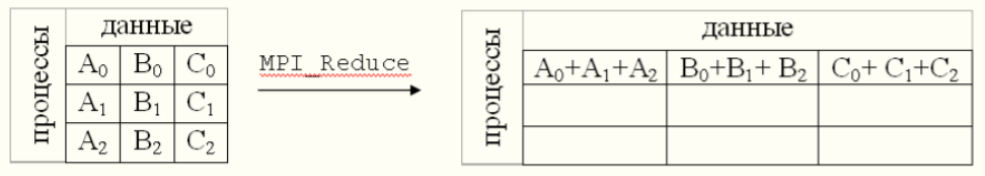
Характерные отличия:

scounts – массив, содержащий количество элементов в каждой части, на которые разбивается сообщение;

displs – массив позиций, определяющий начальные положения каждой части сообщения.

7. int ***MPI_Reduce***(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

Функция проделывает операцию op над данными, хранящимися в sbuf в каждом процессе группы, результат которой записывается в rbuf в процесс с номером root.



Входные параметры:

sbuf – адрес в памяти на каждом процессе, по которому хранятся исходные данные для распределенной операции;

count – количество элементов в sbuf;

datatype – тип данных, над которыми производится распределенная операция;

root – номер процесса, на котором осуществляется размещение результата выполнения операции;

op – название распределенной операции;

comm – коммуникатор.

Выходные параметры:

rbuf – адрес в памяти, по которому размещаются результаты выполнения операции.

12 предопределенных операций:

MPI_MAX – поиск поэлементного максимума;

MPI_MIN – поиск поэлементного минимума;

MPI_SUM – вычисление суммы векторов;

MPI_PROD – вычисление поэлементного произведения векторов;

MPI_LAND – логическое “И”;

MPI_LOR – логическое “ИЛИ”;

MPI_LXOR – логическое исключающее “ИЛИ”;

MPI_BAND – бинарное “И”;

MPI_BOR – бинарное “ИЛИ”;

MPI_BXOR – бинарное исключающее ИЛИ;

MPI_MAXLOC – поиск индексированного максимума;

MPI_MINLOC – поиск индексированного минимума.

Согласно теории конечных разностей:

$$c_{i,j}^n \rho_{i,j}^n \frac{u_{i,j}^{n+1} - u_{ij}^n}{\Delta\tau} = \lambda_{i+\frac{1}{2},j}^n \frac{u_{i+1,j}^n - u_{i,j}^n}{\Delta x^2} - \lambda_{i-\frac{1}{2},j}^n \frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x^2} + \lambda_{i,j+\frac{1}{2}}^n \frac{u_{i,j+1}^n - u_{i,j}^n}{\Delta y^2} - \lambda_{i,j-\frac{1}{2}}^n \frac{u_{i,j}^n - u_{i,j-1}^n}{\Delta y^2},$$

$$\lambda_{i\pm 0.5,j}^n = \lambda\left(\frac{u_{i\pm 1,j}^n - u_{ij}^n}{2}\right), \lambda_{i,j\pm 0.5}^n = \lambda\left(\frac{u_{i,j\pm 1}^n - u_{ij}^n}{2}\right),$$

$$u_{i,j}^0 = u(x, y, 0)$$

$$u_{0,j}^{n+1} = \mu_1(y, \tau),$$

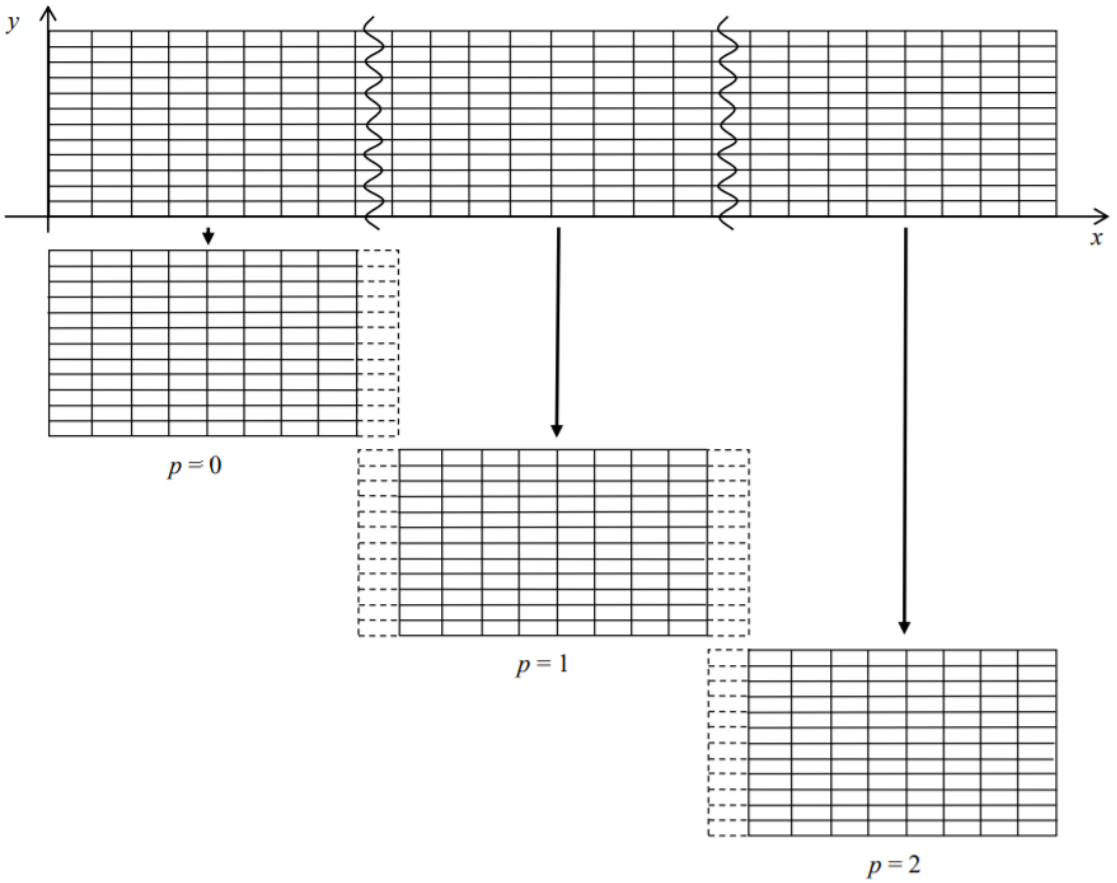
$$u_{i,0}^{n+1} = \mu_3(x, \tau),$$

$$u_{I,j}^{n+1} = \mu_2(y,\tau),$$
$$u_{i,I}^{n+1} = \mu_4(x,\tau),$$

Условие устойчивости:

$$\frac{2\lambda}{c\rho}\Delta\tau \le \left(\frac{1}{\Delta x^2}+\frac{1}{\Delta y^2}\right)^{-1}$$

Декомпозиция пространственной области по оси OX при трех процессорах:



Практическая часть

Таблица 1 - Результаты замеров времени при разном числе процессов (Intel).

p\T_p	I=100, T=80	I=500, T= 0.11	I=1000, T=0.007
1	311,495	276,68	289,727
2	157,73	140,078	143,355
4	79,979	72,972	72,538
8	45,648	37,33	36,294
16	23,909	16,406	19,222
32	14,809	9,703	9,889
64	13,486	5,072	4,706

Таблица 2 - Ускорение при разном числе процессов (Intel)

p\T_p	I=100, T=80	I=500, T= 0.11	I=1000, T=0.007
1	1	1	1
2	1,974862106	1,975185254	2,021045656
4	3,894709861	3,791591295	3,994141002
8	6,823847704	7,41173319	7,982779523
16	13,02835752	16,86456175	15,07267714
32	21,03416841	28,5148923	29,29790677
64	23,09765683	54,55047319	61,56544836

Таблица 3 - Эффективность при разном числе процессов(Intel)

p\T_p	I=100, T=80	I=500, T= 0.11	I=1000, T=0.007
1	1	1	1
2	0,987431053	0,987592627	1,010522828
4	0,973677465	0,947897824	0,99853525
8	0,852980963	0,926466649	0,99784744
16	0,814272345	1,054035109	0,942042321
32	0,657317763	0,891090384	0,915559586
64	0,360900888	0,852351144	0,961960131

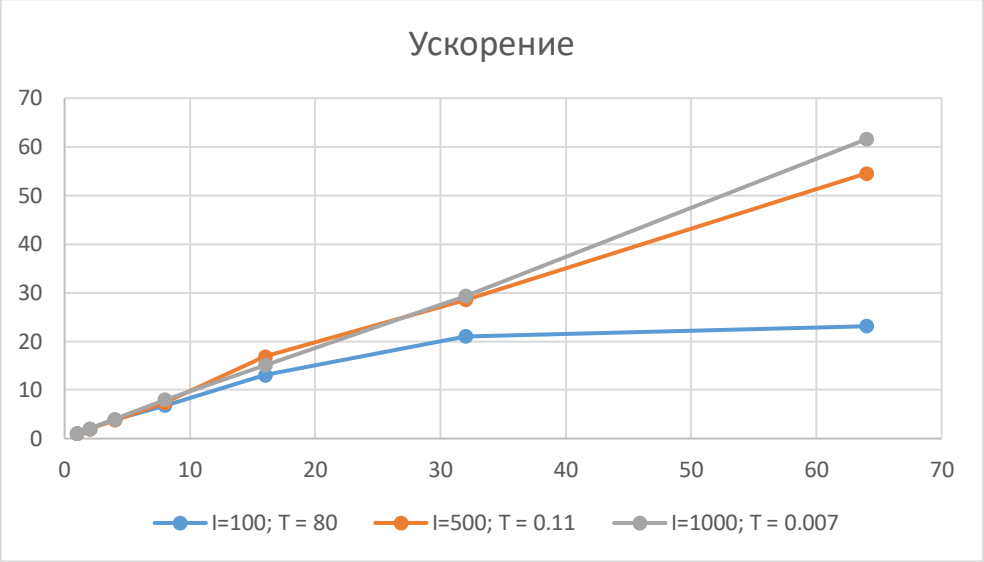


Рисунок 1 - Графики зависимости ускорения от числа процессов при разных размерностях сетки (Intel)

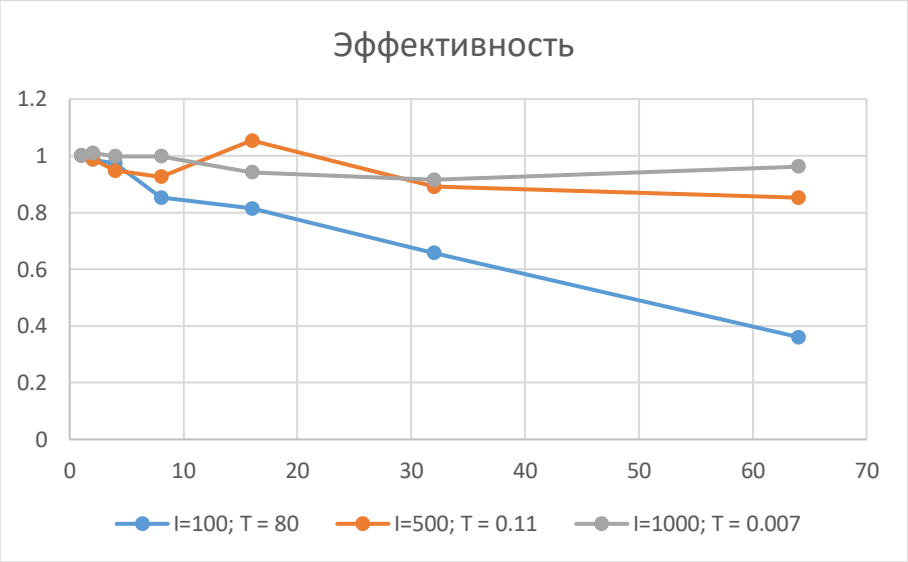


Рисунок 2 - Графики зависимости эффективности от числа процессов при разных размерностях сетки (Intel)

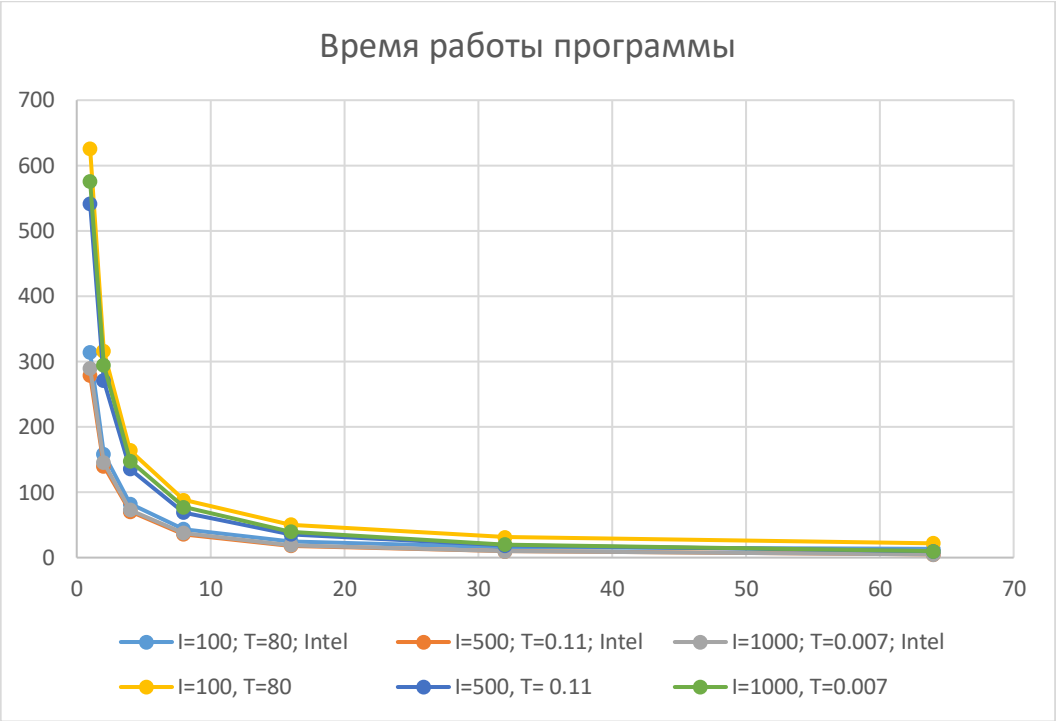


Рисунок 3 – Время работы программы при разных размерностях сетки, для gcc и Intel.

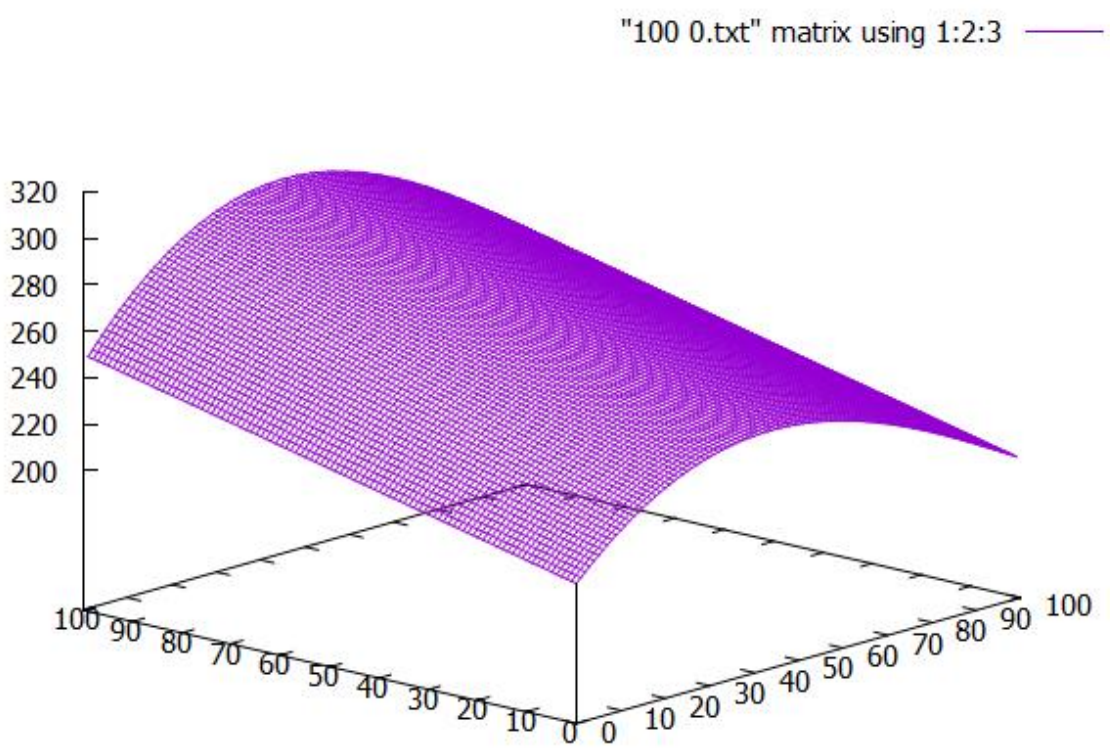


Рисунок 4. Поле температур при T=0 и I=100

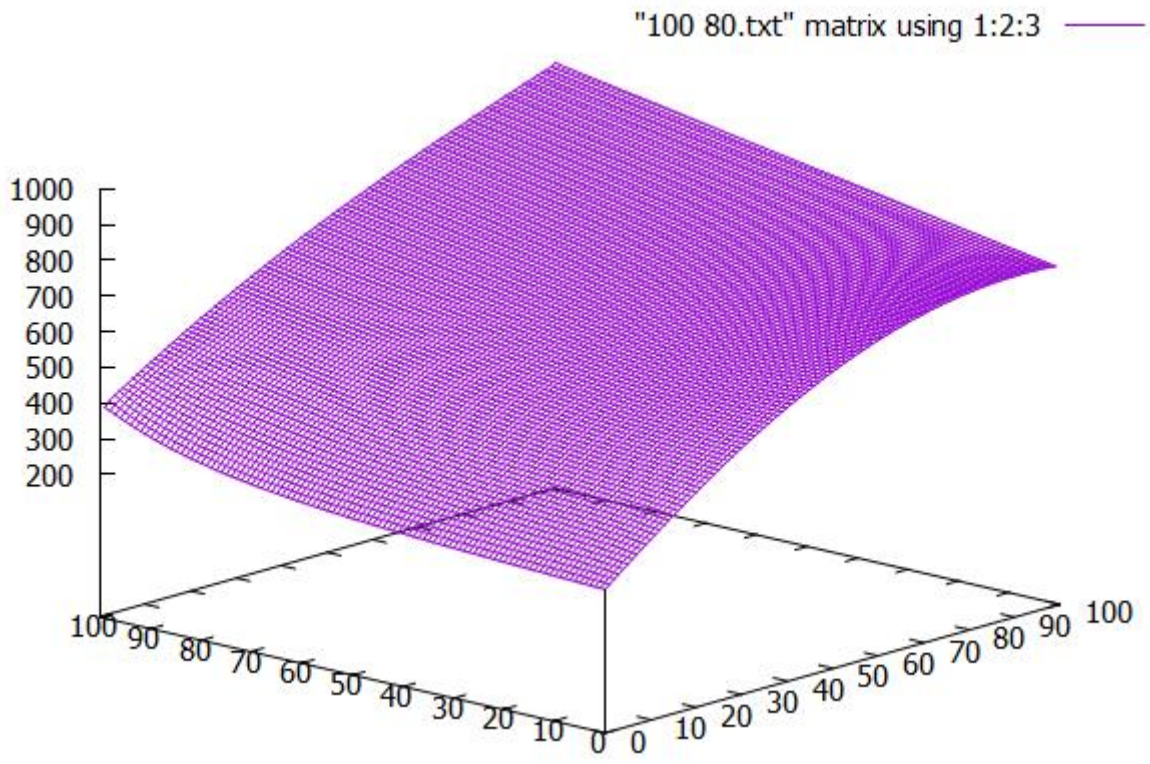


Рисунок 5. Поле температур при T=80 и I=100

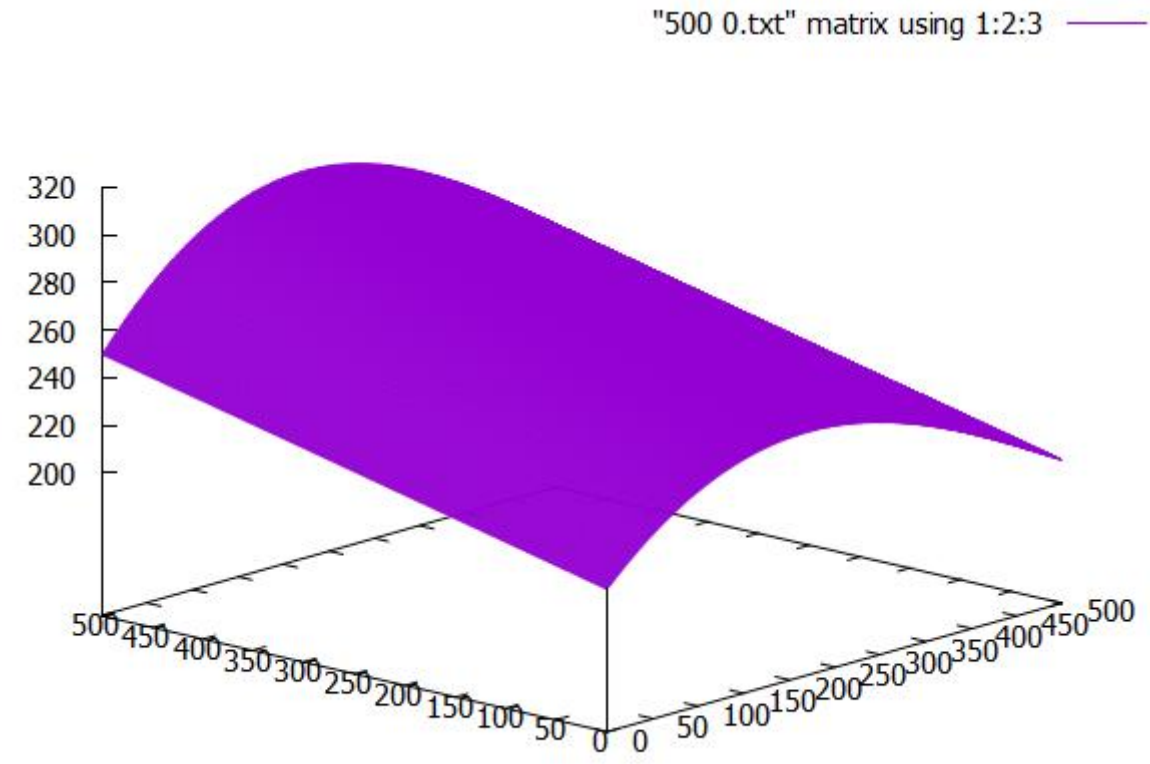


Рисунок 6. Поле температур при T=0 и I=500

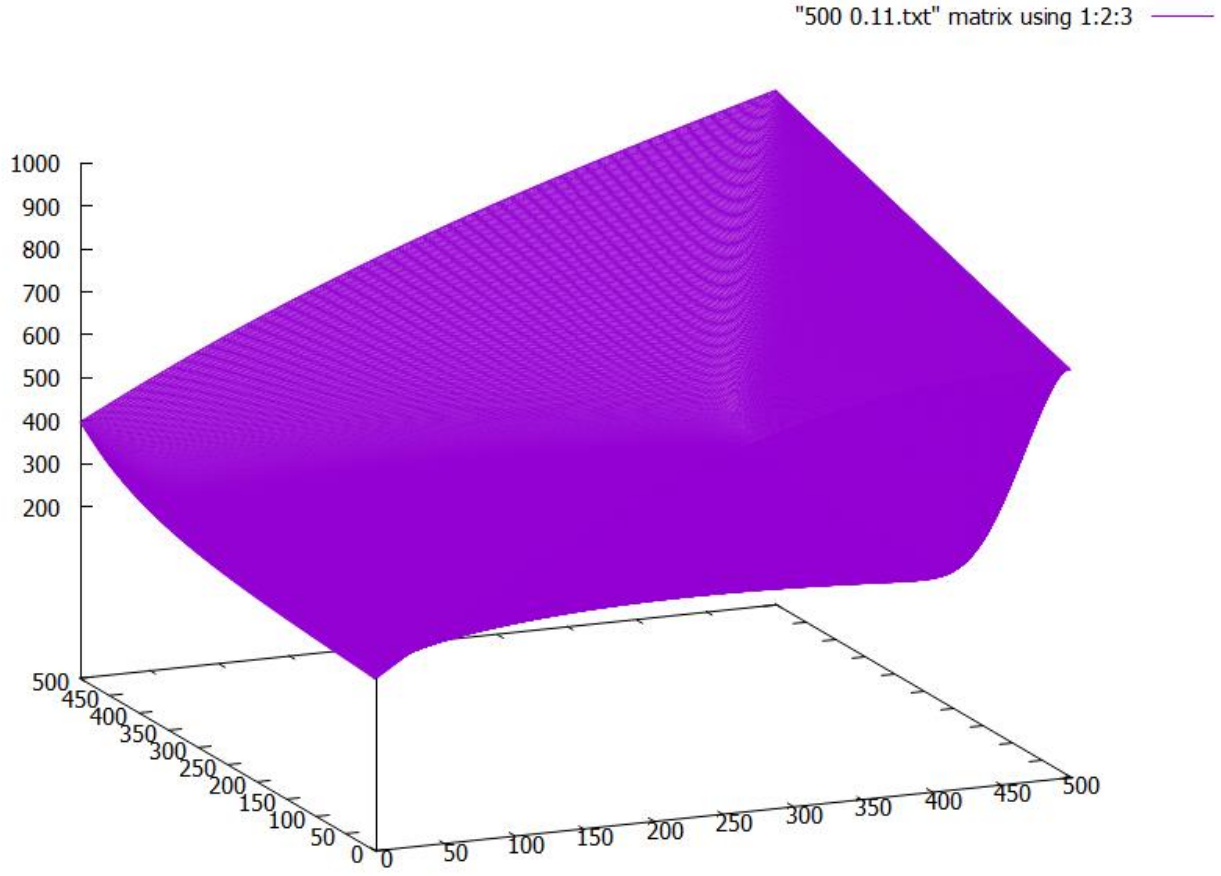


Рисунок 7. Поле температур при T=0.11 и I=500

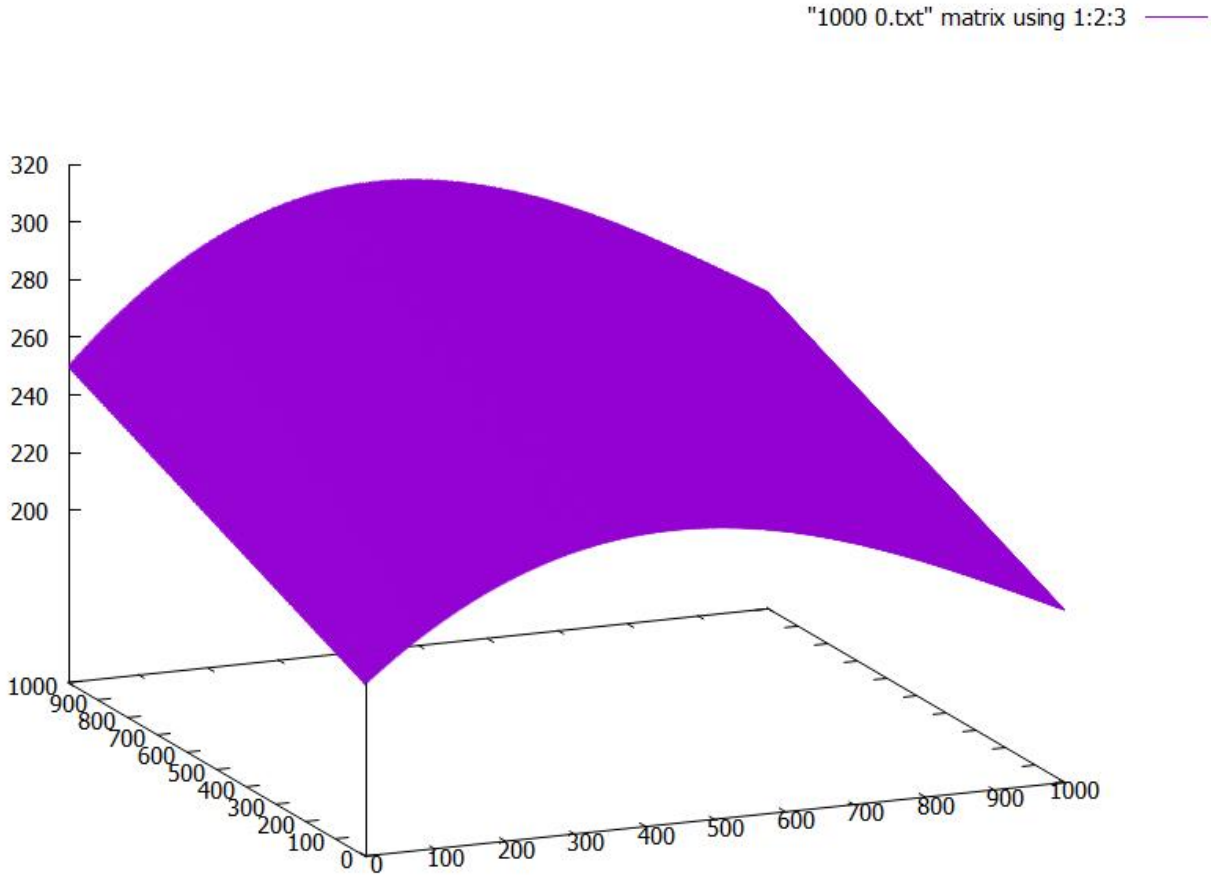


Рисунок 8. Поле температур при T=0 и I=1000

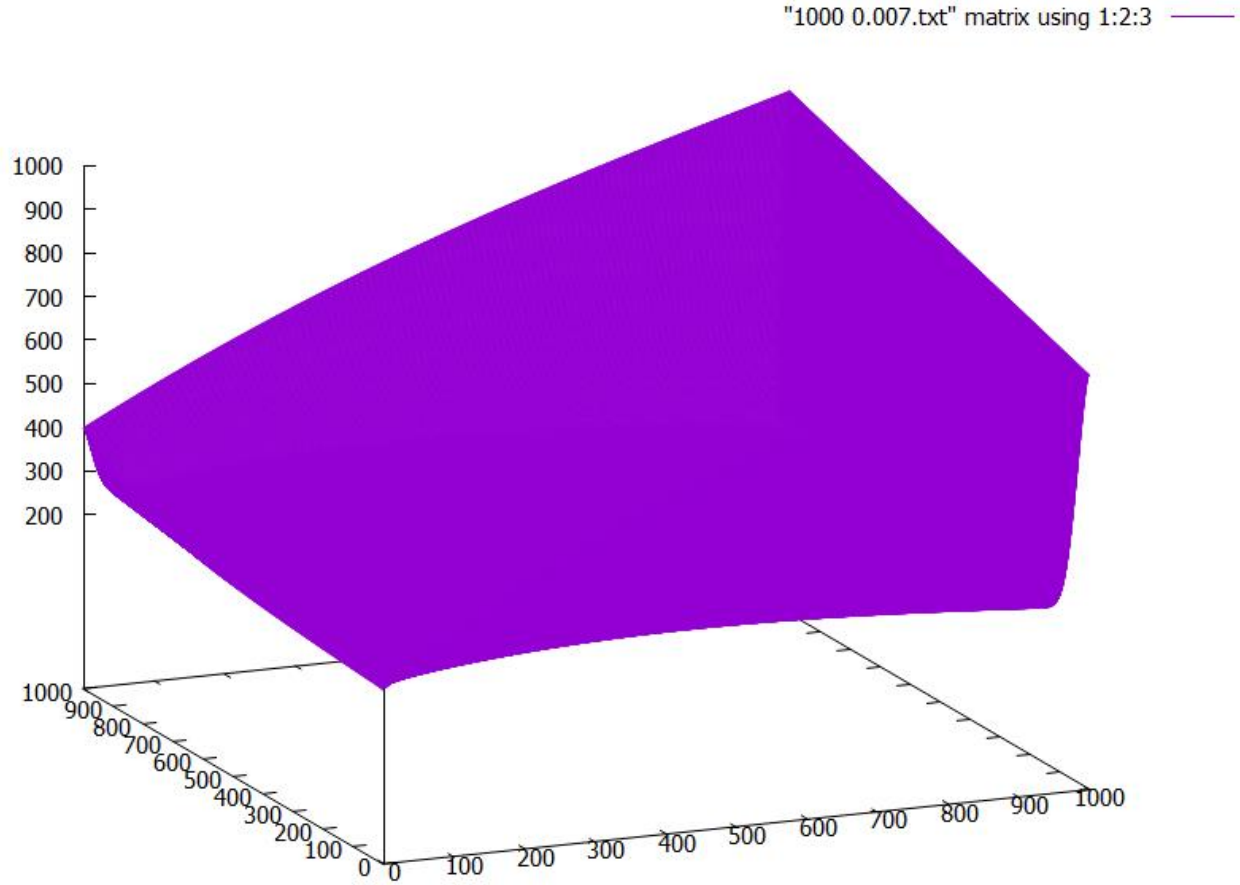


Рисунок 9. Поле температур при T=0.007 и I=1000

Вывод: в ходе лабораторной работы был реализован алгоритм решения двумерного уравнения теплопроводности по явной схеме, проведен анализ его эффективности.

Код программы.

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

#define Pi 3.1415926535897
#define K 0.00001864472034
#define L 0.01 // пространственная область - квадрат с длиной стороны L


using namespace std;

double u0(double x, double y) { return (200. + 50. * y) * (cos(Pi * x / 2.) + x);
}
double mu1(double y, double t) { return 200. + 50. * y + t * exp(5. * y); }
double mu2(double y, double t) { return 200. + 50. * y + t * (550. + 200. * y); }
double mu3(double x, double t) { return 200. * (cos(Pi * x / 2.) + x) + t * (550.
* sin(Pi * x / 2.) + 1. - x); }
double mu4(double x, double t) { return 250. * (cos(Pi * x / 2.) + x) + t * (750.
* x + (1. - x) * exp(5.)); }

double c(double u) { return 1. / (2.25e-3 - 6.08e-10 * u * u); }
double rho(double u) { return 7860. + 41500. / u; }
double lambda(double u) { return 1.45 + 2.3e-2 * u - 2.e-6 * u * u; }


int main(int argc, char* argv[])
{
            int p, o, s = 0.0;
            double* u, * u1;

            int MyID, NumProc, ierror;
            MPI_Status status;
            ierror = MPI_Init(&argc, &argv); //Функция инициализации
            if (ierror != MPI_SUCCESS)
            {
                  printf("MPI initialization error!");
                  exit(1);
            }
            MPI_Comm_size(MPI_COMM_WORLD,   &NumProc);   //Функция   определения
количества процессов в коммуникационной группе с коммуникатором comm
            MPI_Comm_rank(MPI_COMM_WORLD, &MyID); //Функция, возвращающая номер
rank вызвавшего ее процесса, входящего в коммуникационную группу с коммуникатором
comm


            int* rcounts, * displs;
            if (MyID == 0) { rcounts = (int*)calloc(NumProc, sizeof(int)); displs
= (int*)calloc(NumProc, sizeof(int)); }
```

```
            int I = atoi(argv[1]); //I – размер стороны матрицы.
            double T = atof(argv[2]);

            double dx = L / I;
            double tau = (0.9 * dx * dx) / (4.0 * K); //Шаг по времени
            p = (I - 2) / NumProc; //Кол-во операций
            o = (I - 2) % NumProc; //Остатки

            if (MyID < o) { p++; }// пока есть остаток, увеличиваем кол-во
элементов, для 1-го процесса.
            else { s = o; }// иначе s = остатку – то, на сколько элементов
      сдвигаем для следующего блока.


            u = new double[(p + 2) * I]; //(P+2) т.к. выделяется 2 фиктивных
столбца. Размер массива – кол-во элементов для одного процесса.

            if (MyID)
                  u1 = (double*)calloc((p + 2) * I, sizeof(double)); // для
      сборки с каждого процесса, кроме 0-го

            else
            {
                  u1 = (double*)calloc(I * I, sizeof(double)); // общий массив,
куда заносятся все рассчитанные значения.
            }

            for (int j = MyID * p + s; j < (MyID + 1) * p + s + 2; j++)// от
      блока, к блоку. +2 т.к. учитывает фиктивные столбцы.
            {
                  for (int i = 0; i < I; i++)// массив, который хранит начальные
данный для каждого блока.
                  {
                        u[i + (j - MyID * p - s) * I] = u0(i / (double)I, j /
(double)I);
                  }
            }

            if (!MyID)
            {
                  displs[0] = 0; //с какого элемента процесс берет данные
                  for (int i = 0; i < NumProc; i++)
                  {
                        if (i < o) //пока остаток, добавляем столбец
                              rcounts[i] = ((I - 2) / NumProc + 3) * I; //массив
длин принимаемых от процессов сообщений
                        else
                              rcounts[i] = ((I - 2) / NumProc + 2) * I;
                        if (i < NumProc - 1)
                              displs[i + 1] = displs[i] + rcounts[i] - 2 * I;
//массив позиций в приемном буфере, по которым размещаются принимаемые сообщения.
Если не последний блок, тогда указываем номер, с которого передаем данные. -2*I
т.к. в основной матрице нет фиктивных столбцов. Место, с которого передаем данные,
```

зависит от того места, где мы передавали данные предыдущего блока + кол-во элементов в предыдущем блоке, без фиктивных столбцов.

```
                }
            }

        MPI_Gatherv(u, (p + 2) * I, MPI_DOUBLE, u1, rcounts, displs,
      MPI_DOUBLE, 0, MPI_COMM_WORLD);//Функция предназначена для сбора данных со
      всех процессов на одном. Пересылка начальных данных в свои блоки.


            if (MyID == 0)
            {
                  ofstream out("u0.txt");
                  for (int i = 0; i < I; i++)
                  {
                        for (int j = 0; j < I; j++)
                              out << u1[i * I + j] << "\t";
                        out << endl;
                  }
                  out.close();
            }

            double tstart = MPI_Wtime();
            for (double t = tau; t < T; t += tau)
            {
                  for (int j = 1; j < p + 1; j++)
                        for (int i = 1; i < I - 1; i++)
                        {
                              int idx = i + j * I;
                              double lambda1 = lambda((u[idx + 1] + u[idx]) / 2.);
                              double lambda2 = lambda((u[i - 1 + j * I] + u[idx])
/ 2.);
                              double lambda3 = lambda((u[i + (j + 1) * I] + u[idx])
/ 2.);
                              double lambda4 = lambda((u[i + (j - 1) * I] + u[idx])
/ 2.);
                              double cc = c(u[idx]);
                              double roc = rho(u[idx]);
                              u1[idx] = u[idx] + tau / (cc * roc) *
                                    (lambda1 * (u[idx + 1] - u[idx]) / (dx * dx)
                                          - lambda2 * (u[idx] - u[idx - 1]) / (dx *
dx)
                                          + lambda3 * (u[idx + I] - u[idx]) / (dx *
dx)
                                          - lambda4 * (u[idx] - u[idx - I]) / (dx *
dx));
                        }

                  if (MyID)
                        MPI_Sendrecv(&u1[I + 1], I - 2, MPI_DOUBLE, MyID - 1, 1,
&u1[1], I - 2, MPI_DOUBLE, MyID - 1, 1, MPI_COMM_WORLD, &status);//Функция
совмещенного приема/передачи. Для текущего временного слоя отправляем (I-2)
```

элемента с позиции (I+1), тем самым перезаписывая элементы с предыдущего временного слоя (Чтобы не создавать новые массивы). Заполняется только первая строчка.

```
                if (NumProc - MyID - 1)
                      MPI_Sendrecv(&u1[p * I + 1], I - 2, MPI_DOUBLE, MyID + 1,
1, &u1[(p + 1) * I + 1], I - 2, MPI_DOUBLE, MyID + 1, 1, MPI_COMM_WORLD, &status);//
```

Функция совмещенного приема/передачи. На текущем временном слое заполняются оставшиеся ячейки каждого блока.

```
            //Расчет граничных условий для x и y
                if (!MyID)
                      for (int i = 1; i < I - 1; i++)
                      {
                            u1[i] = mu3(i / (double)I, t / T);
                      }

                if (!(NumProc - MyID - 1))
                      for (int i = 1; i < I - 1; i++)
                      {
                            u1[(p + 1) * I + i] = mu4(i / (double)I, t / T);
                      }

                for (int j = 0; j < p + 2; j++)
                {
                      u1[j * I] = mu1((j + MyID * p + s) / (double)I, t / T);
                      u1[(j + 1) * I - 1] = mu2((j + MyID * p + s) / (double)I,
t / T);

                }
            //Расчет граничных условий для x и y
                for (int j = 0; j < p + 2; j++)
                {
                      for (int i = 0; i < I; i++)
                      {
                            u[i + j * I] = u1[i + j * I];
                      }
                }
            }

            MPI_Gatherv(u, (p + 2) * I, MPI_DOUBLE, u1, rcounts, displs,
MPI_DOUBLE, 0, MPI_COMM_WORLD); // Функция предназначена для сбора данных со всех процессов на одном. Собираем посчитанные в единый массив.
            if (MyID == 0)
            {
                double time = MPI_Wtime() - tstart;
                cout << "Time: " << time << endl;
    cout << "NumProc: " << NumProc << endl;
    cout << "I: " << I << endl;
    cout << "T: " << T << endl;
                ofstream out("u1.txt");
                for (int i = 0; i < I; i++)
                {
```

```
                for (int j = 0; j < I; j++)
                      out << u1[i * I + j] << "\t";
                out << endl;
          }
          out.close();
    }

    MPI_Finalize();//функция завершения работы с MPI
    return 0;
}
```