МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Нижегородский государственный университет им. Н. И. Лобачевского»

**Институт информационных технологий, математики и механики**

Направление:

Фундаментальная информатика и информационные технологии

**Отчет**

по лабораторной работе №3

**«Решение дифференциальных уравнений в частных производных. Уравнение Лапласа. Блочная схема»**

**Выполнил:**

студент группы 0836-2

Доронин Роман Олегович

**Проверил:**

Сысоев Александр Владимирович

Нижний Новгород

2016

**Содержание**

[Постановка задачи 1](#_Toc469259352)

[Метод решения 1](#_Toc469259353)

[Схема распараллеливания 1](#_Toc469259354)

[Описание программной реализации 2](#_Toc469259355)

[Подтверждение корректности 2](#_Toc469259356)

[Результаты экспериментов 3](#_Toc469259357)

[Заключение 4](#_Toc469259358)

[Приложение 5](#_Toc469259359)

# Постановка задачи

Дано дифференциальное уравнение в частных производных. Рассмотрим задачу, в которой имеющееся уравнение является уравнением Лапласа. При этом для него задано начальное приближение, то есть значение функции во всех узлах области, а так же граничные условия.

Реализовать последовательный и параллельный вариант решения данной задачи, используя метод релаксации, разделить данные между процессами с помощь блоков, провести сравнительные эксперименты с использованием кластера, оценить эффективность применения параллельных вычислений.

# Метод решения

Для решения уравнения Лапласа

, *(F(x,y) = 0)*

– уравнения эллиптического типа – предназначен метод релаксации. Идея метода заключается в следующем. Если нет источников (F(x,y) = 0), то итоговое значение функции в данном узле на текущем шаге *к+1*, определяется как среднее значение функции в ближайших узлах на предыдущем шаге *к*



Имея начальное приближение, вычисляем примерное решение.

# Схема распараллеливания

Для повышения эффективности вычислений, выполнение операций подсчета приближенных итогового значения функции в узлах сетки, массив, в котором храниться сетка U[i], можно разделить между процессами. Разделив данных на части блоками по процессам, дополнительно потребуется передача друг другу граничных узлов смежных блоков, так как для вычисления значения функции в граничных узлах потребуются, узлы из соседнего блока.

Теперь каждый блок обладает достаточным количеством данных, чтобы вести независимые вычисления итоговых значений во внутренних узлах.

# Описание программной реализации

**Руководство пользователя**

Запуск программы через командную строку осуществляется с помощью команды: mpiexec -n p lab3\_mpi.exe , где p-число процессов, запускаемых для выполнения программы. Начальные условия (начальное приближение) генерируются случайно. Выводятся время работы и сложность параллельного и последовательного алгоритмов, ускорение, разница во времени, результат проверки на корректность.

**Руководство программиста**

Код программы можно просмотреть в разделе «[Приложение](#_Приложение)».

1. Инициализация данных: Начальные условия задаются с помощью генератора случайных чисел на нулевом процессе. Далее происходит копирование исходного массива с целью выполнения последовательного и параллельного алгоритма и последующего сравнения их результатов.
2. Реализация параллельного алгоритма:
   * Создание с помощью **MPI\_Type\_vector** нового типа данныхblock.
   * Заносим сначала на нулевой процесс его блок исходного массива с помощью **MPI\_Sendrecv**. Далее рассылаем на остальные блоки на процессы с помощью **MPI\_Send**.
   * Передаем между смежными блоками граничные узлы.
   * Обработка блоков независимо методом релаксации.
3. Реализация последовательного алгоритма: исполняется на нулевом процессе после параллельного алгоритма, с использование написанной заранее функции Laplas(matr, N, \_itre).
4. После завершения последовательного алгоритма, на нулевом процессе делается сбор данных, выводятся результаты: время, сложность выполнения последовательной части. После завершения параллельного алгоритма, на нулевом процессе делается сбор данных, выводятся результаты: время, сложность выполнения параллельной части. Далее происходит сравнение результатов с целью установления разницы во времени, ускорения и их корректности.

# Подтверждение корректности

Для подтверждения корректности в программе сравниваются массивы, вычисленные последовательным и параллельным методом, и выводится сообщение о том, Correctly (одинаковы) или Faild (не одинаковы). Если результаты не совпали, то работа программы является некорректной.

# Результаты экспериментов

Эксперименты по оценке времени работы алгоритмов, проведены для массивов с размером 100, 200 .. 1000, на 2, 4, 8, 16 процессах.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Время последовательного алгоритма | 2 процесса | | 4 процесса | | 8 процессов | | 16 процессов | |
| время | ускорение | время | ускорение | время | ускорение | время | ускорение |
| 100 | 0,0599 | 0,0356 | 1,6826 | 0,0296 | 2,0212 | 0,0212 | 2,8254 | 0,0194 | 3.0876 |
| 200 | 0,4002 | 0,1838 | 2,1773 | 0,0912 | 4,3868 | 0,0685 | 5,8423 | 0,0351 | 11,4017 |
| 300 | 1,3357 | 0,5069 | 2,635 | 0,2489 | 5,3625 | 0,1875 | 7,1237 | 0,893 | 14,9574 |
| 400 | 3,4522 | 2,8362 | 1,2712 | 2,6059 | 4,0789 | 0,4132 | 8,3547 | 0,2363 | 14,6093 |
| 500 | 5,3726 | 3,2675 | 1,6442 | 1,4641 | 5,3726 | 1,1257 | 4,7726 | 0,5421 | 9,9107 |
| 600 | 9,8602 | 4,2167 | 2,3383 | 2,3375 | 4,2183 | 1,4568 | 6,7683 | 0,625 | 15,7763 |
| 700 | 13,2729 | 6,3605 | 2,0867 | 4,0432 | 4,2828 | 1,5563 | 8,5284 | 0,8712 | 15,2352 |
| 800 | 20,9973 | 9,2631 | 2,1955 | 5,6277 | 4,7310 | 2,502 | 8,3922 | 1,6582 | 12,6627 |
| 900 | 34,3509 | 15,855 | 2,1665 | 10,5137 | 3,2673 | 4,2856 | 8,0154 | 2,2547 | 15,2352 |
| 1000 | 45,0586 | 22,3917 | 2,0123 | 12,1274 | 3,7154 | 5,5241 | 8,1567 | 2,8966 | 15,5557 |

По данным экспериментов видно, что в большинстве случаев распараллеливание привело к уменьшению времени выполнения вычислений, о чем говорит прирост ускорения. Но лишь в тех случаях, когда размер массива много больше количества процессов, на которых выполняются вычисления. Объясняется это тем, что на каждой итерации алгоритма основная операция – выполнение подсчета значения узлов, которая не несёт большой вычислительно нагрузки и выполняется быстро. Но при выполнении распараллеливания время тратиться на раздачу и сбор данных исходного массива между процессами. Поэтому при малых размерностях не стоит запускать выполнение задачи на большом количестве процессов.

# Заключение

Таким образом, в данной лабораторной работе была создана программа, находящая приближенное решение дифференциального уравнения в частных производных. Были реализованы последовательный и параллельный алгоритм с применением блочной схемы, проведены эксперименты по оценке времени работы алгоритмов и ускорения с разными параметрами (количестве процессов и размер исходного массива).

# Приложение

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include <ctime>

#include "mpi.h"

#include <iostream>

#include <windows.h>

enum ConsoleColor

{

Black = 0,

Blue = 1,

Green = 2,

Cyan = 3,

Red = 4,

Magenta = 5,

Brown = 6,

LightGray = 7,

DarkGray = 8,

LightBlue = 9,

LightGreen = 10,

LightCyan = 11,

LightRed = 12,

LightMagenta = 13,

Yellow = 14,

White = 15

};

void SetColor(ConsoleColor text, ConsoleColor background)

{

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hStdOut, (WORD)((background << 4) | text));

}

void Laplas(float \*matr, int N, int &\_iter)

{

float eps = 0.1;

float dmax, temp, dm;

int elem\_count = (N + 1)\*(N + 1);

float \*u = new float[elem\_count];

for (int i = 0; i < elem\_count; i++)

u[i] = matr[i];

float ctrl;

do {

dmax = 0; // максимальное изменение значений u

for (int i = 1; i < (N); i++)

for (int j = 1; j < (N); j++) {

temp = u[i \* (N) + j];

ctrl = 0.25\*(u[(i - 1)\*(N)+j] + u[(i + 1)\*(N)+j] + u[i\*(N)+(j - 1)] + u[i\*(N)+(j + 1)]);

u[i\*(N)+j] = 0.25\*(u[(i - 1)\*(N)+j] + u[(i + 1)\*(N)+j] + u[i\*(N)+(j - 1)] + u[i\*(N)+(j + 1)]);

dm = fabs(temp - u[i\*(N)+j]); // abs

if (dmax < dm) dmax = dm;

}

\_iter++;

} while (dmax > eps);

}

int main(int argc, char\* argv[])

{

int proc\_num, proc\_rank;

int chunk\_size;

double parallel\_start, parallel\_end;

double linear\_start, linear\_end;

int iter = 0, yo = 0;

float eps = 0.1;

int N = 9999;

float ftemp, dm;

int elem\_count = (N + 1)\*(N + 1);

float \*u = new float[elem\_count];

float \*peredacha = 0;

//---------------------------------------Начало---------------------------------------

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);

float dmax = 0;

int NB = sqrt(proc\_num); //!!!!!!!!!!!!!!!!!!!!!!

chunk\_size = (N + 1) / NB;

// Создание bufs для граничных точек

float \*top = new float[chunk\_size], \*left = new float[chunk\_size], \*bot = new float[chunk\_size], \*right = new float[chunk\_size];

if (proc\_rank == 0)

{

std::cout << "Chunk\_Size = " << chunk\_size << std::endl;

peredacha = new float[elem\_count];

for (int i = 0; i < elem\_count; i++)

u[i] = rand() % (N \* 2) - N;

srand(MPI\_Wtime() \* 1000);

for (int i = 0; i < elem\_count; i++)

peredacha[i] = u[i];

parallel\_start = MPI\_Wtime();

}

float \*chunk = new float[chunk\_size\*chunk\_size];

MPI\_Datatype block; // Особый тип данных

MPI\_Type\_vector(

chunk\_size, // Число блоков

chunk\_size, // Число элементов базового типа в каждом блоке

N + 1, // Шаг между началами соседних блоков, измеренный числом элементов базового типа

MPI\_FLOAT, // Базовый тип данных

&block); // Новый производный тип данных

MPI\_Type\_commit(&block); // Регисрация нового произвольного типа

MPI\_Status status;

if (proc\_rank == 0) // Заносим в chunk массив блоков на 0 процесс

// posilaem i prinimaem chast v 0

MPI\_Sendrecv(

&u[0], 1, block, 0, 8, // Параметры передаваемого сообщения

chunk, chunk\_size\*chunk\_size, MPI\_FLOAT, 0, 8, // Параметры принимаемого сообщения

MPI\_COMM\_WORLD, &status);

if (proc\_rank == 0) // Рассылаем по блокам на процессы // rassilaem vsem ostalnim

for (int s = 1; s < proc\_num; s++)

MPI\_Send(&u[((s / NB)\*(N + 1) + (s % NB))\*chunk\_size], 1, block, s, 9, MPI\_COMM\_WORLD);

if (proc\_rank != 0) // Принимаем

MPI\_Recv(chunk, chunk\_size\*chunk\_size, MPI\_FLOAT, 0, 9, MPI\_COMM\_WORLD, &status);

float i1, i2, j1, j2;

int ek = 0, el = 0, lk = 0, ll = 0;

if (proc\_rank / NB == 0) ek = 1;

if (proc\_rank % NB == 0) el = 1;

if (proc\_rank / NB == NB - 1) lk = -1;

if (proc\_rank % NB == NB - 1) ll = -1;

float fer;

do {

dmax = 0;

// Получение граничных узлов

if (proc\_rank / NB != 0) { // Строка не нулевая

// Получение данных от верхнего процессора

MPI\_Status status;

MPI\_Recv(top, chunk\_size, MPI\_FLOAT, proc\_rank - NB, 10, MPI\_COMM\_WORLD, &status);

// Пересылка данных verhnemy процессору

// Verhnyaa строка

float \*temp = new float[chunk\_size];

for (int i = 0; i < chunk\_size; i++)

temp[i] = chunk[i]; // Посылка строки

MPI\_Send(temp, chunk\_size, MPI\_FLOAT, proc\_rank - NB, 1, MPI\_COMM\_WORLD);

}

if (proc\_rank % NB != 0) { // столбец не нулевой

//prinyat ot levogo

MPI\_Status status;

int k = MPI\_Recv(left, chunk\_size, MPI\_FLOAT, proc\_rank - 1, 2, MPI\_COMM\_WORLD, &status);

// пересылка pravogo данных levomy процессору

float \*temp = new float[chunk\_size];

for (int i = 0; i < chunk\_size; i++)

temp[i] = chunk[i \* chunk\_size]; // Посылка столбца

MPI\_Send(temp, chunk\_size, MPI\_FLOAT, proc\_rank - 1, 3, MPI\_COMM\_WORLD);

}

// пересылка граничных узлов

if (proc\_rank / NB != NB - 1) {

// пересылка данных verhnemu процессору

float \*temp = new float[chunk\_size];

for (int i = 0; i < chunk\_size; i++)

temp[i] = chunk[chunk\_size\*(chunk\_size - 1) + i];

MPI\_Send(temp, chunk\_size, MPI\_FLOAT, proc\_rank + NB, 10, MPI\_COMM\_WORLD);

MPI\_Status status;

int k = MPI\_Recv(bot, chunk\_size, MPI\_FLOAT, proc\_rank + NB, 1, MPI\_COMM\_WORLD, &status);

}

if (proc\_rank % NB != NB - 1) {

//pravomu pravoe

float \*temp = new float[chunk\_size];

for (int i = 0; i<chunk\_size; i++)

temp[i] = chunk[i\*chunk\_size + chunk\_size - 1];

MPI\_Send(temp, chunk\_size, MPI\_FLOAT, proc\_rank + 1, 2, MPI\_COMM\_WORLD);

MPI\_Status status;

int k = MPI\_Recv(right, chunk\_size, MPI\_FLOAT, proc\_rank + 1, 3, MPI\_COMM\_WORLD, &status);

}

// Обработка блока с оценкой погрешности dmax

for (int i = ek; i < chunk\_size + lk; i++)

for (int j = el; j < chunk\_size + ll; j++) {

ftemp = chunk[i \* (chunk\_size) + j];

// Вычисление соседей

if (i - 1 >= 0) i1 = chunk[(i - 1) \* (chunk\_size) + j];

else i1 = top[j];

if (i + 1 < chunk\_size) i2 = chunk[(i + 1) \* (chunk\_size) + j];

else i2 = bot[j];

if (j - 1 >= 0) j2 = chunk[i \* (chunk\_size) + (j - 1)];

else j2 = left[i];

if (j + 1 < chunk\_size) j1 = chunk[i \* (chunk\_size) + (j + 1)];

else j1 = right[i];

chunk[i \* (chunk\_size) + j] = 0.25 \* (i1 + i2 + j1 + j2);

dm = fabs(ftemp - chunk[i\*(chunk\_size)+j]); // abs

if (dmax < dm) dmax = dm;

}

iter++;

// синхронизация и рассылка погрешности dmax

MPI\_Barrier(MPI\_COMM\_WORLD);

MPI\_Reduce(&dmax, &fer, 1, MPI\_FLOAT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

dmax = fer;

MPI\_Bcast(&dmax, 1, MPI\_FLOAT, 0, MPI\_COMM\_WORLD);

} while (dmax > eps);

if (proc\_rank == 0)

{

parallel\_end = MPI\_Wtime();

std::cout << std::endl << "-----Parallel version-----" << std::endl;

std::cout << "Time: " << parallel\_end - parallel\_start << " sec" << std::endl;

std::cout << "The number of operation: " << iter << std::endl << std::endl;

linear\_start = MPI\_Wtime();

Laplas(peredacha, N, iter);

linear\_end = MPI\_Wtime();

std::cout << std::endl << "-----Linear version-----" << std::endl;

std::cout << "Time: " << linear\_end - linear\_start << " sec" << std::endl;

std::cout << "The number of operation: " << iter << std::endl << std::endl << std::endl;

std::cout << "The time difference: "

<< linear\_end - linear\_start - parallel\_end + parallel\_start

<< std::endl;

std::cout << "Acceleration: "

<< (linear\_end - linear\_start) / (parallel\_end - parallel\_start)

<< std::endl << std::endl;

int flag = 1;

for (int i = 0; i < chunk\_size\*chunk\_size; i++)

if (chunk[i] != peredacha[i]) flag = 0;

std::cout << "Check for correctness: ";

if (flag) { std::cout << "["; SetColor(Red, Black); std::cout << " Faild "; SetColor(White, Black); std::cout << "]" << std::endl << std::endl; }

else { std::cout << "["; SetColor(LightGreen, Black); std::cout << " Correctly "; SetColor(White, Black); std::cout << "]" << std::endl << std::endl; }

}

MPI\_Finalize();

return 0;

}