МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ) Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №2

по курсу «Параллельная обработка данных»

Технологии MPI и OpenMP

Выполнила: Алексюнина Ю.В.

Группа:М8О-407Б

Преподаватели:

К.Г.Крашенинников, А.Ю. Морозов

Москва, 2021

Условие:

Цель работы: Совместное использование технологии MPI и технологии OpenMP. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

Вариант: 1. Распараллеливание основных циклов через parallel for (+директива reduction для вычисления погрешности)

Программное и аппаратное обеспечение

GPU:
- Name: GeForce GTX 750 Ti
- Compute capability: 5.0
- Графическая память: 4294967295
- Разделяемая память: 49152
- Константная память: 65536
- Количество регистров на блок: 65536
- Максимальное количество блоков: (2147483647, 65535, 65535)
- Максимальное количество нитей: (1024, 1024, 64)
- Количество мультипроцессоров: 5

Сведения о системе:
- Процессор: Intel Core i5-4460 3.20GHz
- ОЗУ: 16 ГБ
- HDD: 930 ГБ

Программное обеспечение:
- OS: Windows 8.1
- IDE: Visual Studio 2019

Метод решения:

Для выполнения данной лабораторной работы необходима схема решения из лабораторной работы № 1. Однако второй этап можно распараллелить на каждом из процессов (каждый цикл с принятием данных и перерасчетом) с помощью технологии OpenMP. Каждый процесс в потоке будет перерассчитывать значения для отдельного участка памяти в блоке.

Схема решения:

1. Передать данные другим процессам на границах.

2. Обновить данные во всех ячейках.

3. Вычисление локальной (в рамках процесса) погрешности и во всей области.

Описание программы:

В отличие от лабораторной № 1 добавились директивы препроцессора #pragma parallel, которые позволяют задавать участки, которые будут выполнятся в многопроцессорном режиме для каждого из потоков. Для того, чтобы каждый поток отвечал за отдельный участок, нужно было добавить эту директиву перед каждым циклом, за исключением вычисления погрешности.
#pragma omp parallel for private(i, j, k) shared(data, edge_yz)
Где edge_yz - буффер для конкретной области, а data - массив с данными.
#pragma omp parallel for private(i, j, k) shared(data, next) reduction(max: difference)
Директива reduction необходима для вычисления погрешности.

Файл lab9.cpp:

```
//#pragma warning(disable : 4996)


#include <algorithm>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
#include <iostream>
#include <cmath>
#include <omp.h>
using namespace std;


// Индексация внутри блока
#define _i(i, j, k) (((k) + 1) * (dim.y + 2) * (dim.x + 2) + ((j) + 1) * (dim.x + 2) + (i) + 1)


// Индексация по блокам (процессам)
#define _ib(i, j, k) ((k) * block.y * block.x + (j) * block.x + (i))
#define _ibz(id) ((id) / block.y / block.x)
#define _iby(id) (((id) % (block.y * block.x)) / block.x)
```

```
#define _ibx(id) ((id) % block.x)

struct
{
    int x;
    int y;
    int z;
} dim;

struct
{
    int x;
    int y;
    int z;
} block;

struct
{
    double x;
    double y;
    double z;
} l; //area size

struct
{
    double down;
    double up;
    double left;
    double right;
    double front;
    double back;

} u; //border conditions
```

```
int main(int argc, char* argv[])
{
    int id; //номер процесса, вызвавшего функцию
    int bx, by, bz;
    int i, j, k;
    int numproc, proc_name_len; //число процессов
    char proc_name[MPI_MAX_PROCESSOR_NAME];
    double eps, u0;
    double* temp;
    string outFile;
    double diff, total_diff = 0;

    MPI_Status status; //статус выполнения операций mpi
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numproc); //число процессов в
области связи коммуникатора comm
    MPI_Comm_rank(MPI_COMM_WORLD, &id); //номер процесса, вызвавшего
функцию
    MPI_Get_processor_name(proc_name, &proc_name_len);


    if (id == 0) //если главный процесс
    {
        //input data
        cin >> block.x >> block.y >> block.z;          // Размер сетки
блоков (процессов)
        cin >> dim.x >> dim.y >> dim.z; // Размер блока
        cin >> outFile;
        cin >> eps;
        cin >> l.x >> l.y >> l.z;
        cin >> u.down >> u.up >> u.left >> u.right >> u.front >>
u.back;
        cin >> u0;
    }
```

```
    // Передача параметров расчета всем процессам
    MPI_Bcast(&dim, 3, MPI_INT, 0, MPI_COMM_WORLD); //Процесс с номером root(0) рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора
    MPI_Bcast(&block, 3, MPI_INT, 0, MPI_COMM_WORLD); //то есть, передаем 3 эл-та из block всем процессам
    MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&l, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&u, 6, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&u0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);



    //buf init
    double* data = (double*)malloc(sizeof(double) * (dim.x + 2) * (dim.y + 2) * (dim.z + 2));
    double* next = (double*)malloc(sizeof(double) * (dim.x + 2) * (dim.y + 2) * (dim.z + 2));
    double* buff = (double*)malloc(sizeof(double) * (max(dim.x, max(dim.y, dim.z)) * max(dim.x, max(dim.y, dim.z)) + 2));
    double* edge_xy = (double*)malloc(sizeof(double) * dim.x * dim.y);
    double* edge_xz = (double*)malloc(sizeof(double) * dim.x * dim.z);
    double* edge_yz = (double*)malloc(sizeof(double) * dim.y * dim.z);


    //zero iteration
    for (i = 0; i < dim.x; i++)
    {
        for (j = 0; j < dim.y; j++)
        {
            for (k = 0; k < dim.z; k++)
            {
                data[_i(i, j, k)] = u0;
            }

        }

    }
```

```
    bx = _ibx(id);    // Переход к 3-мерной индексации процессов
    by = _iby(id);
    bz = _ibz(id);

    double hx = l.x / (dim.x * block.x);
    double hy = l.y / (dim.y * block.y);
    double hz = l.z / (dim.z * block.z);

    omp_set_num_threads(1); //2 threads for speed

    do
    {
        if (bx < block.x - 1)
        {
            for (k = 0; k < dim.z; k++)
            {
                for (j = 0; j < dim.y; j++)
                {
                    //buff[k * dim.y + j] = data[_i(dim.x - 1, j, k)];
                    edge_yz[k * dim.y + j] = data[_i(dim.x - 1, j,
k)];
                }
            }
            MPI_Send(edge_yz, dim.y * dim.z, MPI_DOUBLE, _ib(bx + 1,
by, bz), id, MPI_COMM_WORLD);
        }

        if (bx > 0)
        {
            MPI_Recv(edge_yz, dim.y * dim.z, MPI_DOUBLE, _ib(bx - 1,
by, bz), _ib(bx - 1, by, bz), MPI_COMM_WORLD, &status);
            #pragma omp parallel for private(i, j, k) shared(data,
edge_yz)
```

```
            for (k = 0; k < dim.z; k++)
            {
                for (j = 0; j < dim.y; j++)
                {
                    data[_i(-1, j, k)] = edge_yz[k * dim.y + j];
//buff[k * dim.y + j];
                }
            }

        }
        else
        {
            #pragma omp parallel for private(i, j, k) shared(data)
            for (k = 0; k < dim.z; k++)
            {
                for (j = 0; j < dim.y; j++)
                {
                    data[_i(-1, j, k)] = u.left;
                }
            }
        }

        if (by < block.y - 1)
        {
            for (k = 0; k < dim.z; k++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    //buff[k * dim.x + i] = data[_i(i, dim.y - 1, k)];
                    edge_xz[k * dim.x + i] = data[_i(i, dim.y - 1,
k)];
                }
            }
            MPI_Send(edge_xz, dim.z * dim.x, MPI_DOUBLE, _ib(bx, by +
1, bz), id, MPI_COMM_WORLD);
```

```
        }


        if (by > 0)
        {
            MPI_Recv(edge_xz, dim.x * dim.z, MPI_DOUBLE, _ib(bx, by -
1, bz), _ib(bx, by - 1, bz), MPI_COMM_WORLD, &status);
            #pragma omp parallel for private(i, j, k) shared(data,
edge_xz)
            for (k = 0; k < dim.z; k++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    data[_i(i, -1, k)] = edge_xz[k * dim.x + i];
//buff[k * dim.x + i];
                }


            }


        }
        else
        {
            #pragma omp parallel for private(i, j, k) shared(data)
            for (k = 0; k < dim.z; k++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    data[_i(i, -1, k)] = u.front;
                }
            }
        }


        if (bz < block.z - 1)
        {
            for (j = 0; j < dim.y; j++)
            {
```

```
                for (i = 0; i < dim.x; i++)
                {
                    edge_xy[j * dim.x + i] = data[_i(i, j, dim.z - 1)];
                    //buff[j * dim.x + i] = data[_i(i, j, dim.z - 1)];
                    //printf("%f\n", buff[i + j * dim.x]);
                }
            }
            //double r = _ib(bx, by, bz + 1);
            //cout << r;
            MPI_Send(edge_xy, dim.y * dim.x, MPI_DOUBLE, _ib(bx, by, bz + 1), id, MPI_COMM_WORLD);
        }

        if (bz > 0)
        {
            MPI_Recv(edge_xy, dim.x * dim.y, MPI_DOUBLE, _ib(bx, by, bz - 1), _ib(bx, by, bz - 1), MPI_COMM_WORLD, &status);
            #pragma omp parallel for private(i, j, k) shared(data, edge_xy)
            for (j = 0; j < dim.y; j++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    data[_i(i, j, -1)] = edge_xy[j * dim.x + i]; //buff[j * dim.x + i];
                }
            }

        }
        else
        {
            #pragma omp parallel for private(i, j, k) shared(data)
            for (j = 0; j < dim.y; j++)
            {
                for (i = 0; i < dim.x; i++)
```

```
                {
                    data[_i(i, j, -1)] = u.down;
                }
            }
        }

        if (bx > 0)
        {
            for (k = 0; k < dim.z; k++)
            {
                for (j = 0; j < dim.y; j++)
                {
                    edge_yz[k * dim.y + j] = data[_i(0, j, k)];
                    //buff[k * dim.y + j] = data[_i(0, j, k)];
                }
            }
            MPI_Send(edge_yz, dim.z * dim.y, MPI_DOUBLE, _ib(bx - 1,
by, bz), id, MPI_COMM_WORLD);
        }

        if (bx < block.x - 1)
        {
            MPI_Recv(edge_yz, dim.y * dim.z, MPI_DOUBLE, _ib(bx + 1,
by, bz), _ib(bx + 1, by, bz), MPI_COMM_WORLD, &status);
            #pragma omp parallel for private(i, j, k) shared(data,
edge_yz)
            for (k = 0; k < dim.z; k++)
            {
                for (j = 0; j < dim.y; j++)
                {
                    data[_i(dim.x, j, k)] = edge_yz[k * dim.y + j];
//buff[k * dim.y + j];
                }
            }
        }
```

```
        else
        {
            #pragma omp parallel for private(i, j, k) shared(data)
            for (k = 0; k < dim.z; k++)
            {
                for (j = 0; j < dim.y; j++)
                {
                    data[_i(dim.x, j, k)] = u.right;
                }
            }
        }

        if (by > 0)
        {
            for (k = 0; k < dim.z; k++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    edge_xz[k * dim.x + i] = data[_i(i, 0, k)];
                    //buff[k * dim.x + i] = data[_i(i, 0, k)];
                }
            }
            MPI_Send(edge_xz, dim.z * dim.x, MPI_DOUBLE, _ib(bx, by -
1, bz), id, MPI_COMM_WORLD);
        }

        if (by < block.y - 1)
        {
            MPI_Recv(edge_xz, dim.x * dim.z, MPI_DOUBLE, _ib(bx, by +
1, bz), _ib(bx, by + 1, bz), MPI_COMM_WORLD, &status);
            #pragma omp parallel for private(i, j, k) shared(data,
edge_xz)
            for (k = 0; k < dim.z; k++)
            {
                for (i = 0; i < dim.x; i++)
```

```
                {
                    data[_i(i, dim.y, k)] = edge_xz[k * dim.x + i];
                }
            }
        }
        else
        {
            #pragma omp parallel for private(i, j, k) shared(data)
            for (k = 0; k < dim.z; k++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    data[_i(i, dim.y, k)] = u.back;
                }
            }

        }

        if (bz > 0)
        {
            for (j = 0; j < dim.y; j++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    edge_xy[j * dim.x + i] = data[_i(i, j, 0)];
                }
            }
            MPI_Send(edge_xy, dim.y * dim.x, MPI_DOUBLE, _ib(bx, by,
bz - 1), id, MPI_COMM_WORLD);
        }

        if (bz < block.z - 1)
        {
            MPI_Recv(edge_xy, dim.x * dim.y, MPI_DOUBLE, _ib(bx, by,
bz + 1), _ib(bx, by, bz + 1), MPI_COMM_WORLD, &status);
```

```
            #pragma omp parallel for private(i, j, k) shared(data,
edge_xy)
            for (j = 0; j < dim.y; j++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    data[_i(i, j, dim.z)] = edge_xy[i + j * dim.x];
                }
            }
        }
        else
        {
            #pragma omp parallel for private(i, j, k) shared(data)
            for (j = 0; j < dim.y; j++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    data[_i(i, j, dim.z)] = u.up;
                }
            }
        }

        diff = 0.0;
//ыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыыы

        #pragma omp parallel for private(i, j, k) shared(data, next,
hx, hy, hz)
        for (i = 0; i < dim.x; i++)
        {
            for (j = 0; j < dim.y; j++)
            {
                for (k = 0; k < dim.z; k++)
                {
                    next[_i(i, j, k)] = ((data[_i(i + 1, j, k)] +
data[_i(i - 1, j, k)]) / (hx * hx) +
```

```
                        (data[_i(i, j + 1, k)] + data[_i(i, j - 1,
k)]) / (hy * hy) +
                        (data[_i(i, j, k + 1)] + data[_i(i, j, k -
1)]) / (hz * hz)) /
                        (2 * (1.0 / (hx * hx) + 1.0 / (hy * hy) + 1.0
/ (hz * hz)));
                    //diff = max(diff, fabs(next[_i(i, j, k)] -
data[_i(i, j, k)]));
                }
            }
        }

        #pragma omp parallel for private(i, j, k) shared(data, next)
reduction(max: diff)
        for (i = 0; i < dim.x; i++)
        {
            for (j = 0; j < dim.y; j++)
            {
                for (k = 0; k < dim.z; k++)
                {
                    diff = max(diff, fabs(next[_i(i, j, k)] -
data[_i(i, j, k)]));
                }
            }
        }

        temp = next;
        next = data;
        data = temp;

        total_diff = 0.0;
        double* diffs = (double*)malloc(sizeof(double) * block.x *
block.y * block.z);
        MPI_Allgather(&diff, 1, MPI_DOUBLE, diffs, 1, MPI_DOUBLE,
MPI_COMM_WORLD);

        for (k = 0; k < block.x * block.y * block.z; k++)
```

```
        {
            total_diff = max(total_diff, diffs[k]);
        }



    } while (total_diff > eps); //ыыыыыыыыыыыыыыыыы x2


    if (id != 0)
    {
        for (k = 0; k < dim.z; k++)
        {
            for (j = 0; j < dim.y; j++)
            {
                for (i = 0; i < dim.x; i++)
                {
                    buff[i] = data[_i(i, j, k)];
                }
                MPI_Send(buff, dim.x, MPI_DOUBLE, 0, id,
MPI_COMM_WORLD);
            }
        }
    }
    else
    {
        FILE* fd;
        fd = fopen(outFile.c_str(), "w");
        for (bz = 0; bz < block.z; bz++)
        {
            for (k = 0; k < dim.z; k++)
            {
                for (by = 0; by < block.y; by++)
                {
                    for (j = 0; j < dim.y; j++)
```

```
                    {
                        for (bx = 0; bx < block.x; bx++)
                        {
                            if (_ib(bx, by, bz) == 0)
                            {
                                for (i = 0; i < dim.x; i++)
                                {
                                    buff[i] = data[_i(i, j, k)];
                                }
                            }
                            else
                            {
                                MPI_Recv(buff, dim.x, MPI_DOUBLE,
_ib(bx, by, bz), _ib(bx, by, bz), MPI_COMM_WORLD, &status);
                            }

                            for (i = 0; i < dim.x; i++)
                            {
                                fprintf(fd, "%.7e ", buff[i]);
                            }
                        }
                    }
                }
            }
        }
        fclose(fd);
    }

    MPI_Finalize();

    free(buff);
    free(data);
    free(next);
```

```
    return 0;
}
```

Результаты:

	MPI+ Open MP	MPI	CPU
1 1 1, 20 20 20	**7562ms**	**18933ms**	**10347ms**

Выводы:

Выполнение данной лабораторной работы показывает, что параллельная обработка данных с несколькими потоками, внутри которых находятся несколько процессов, происходит гораздо быстрее. Технология Open MP, позволяет распараллеливать код с помощью легковесных потоков достаточно легко и быстро. Ее использование вместе с MPI позволяет разбивать программу на процессы, каждый из которых будет иметь несколько потоков исполнения и получать существенный прирост в скорости вычисления.