МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №7
по курсу «Программирование графических процессоров»

Message Passing Interface (MPI)

Выполнил: Полей-Добронравова Амелия
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников, А.Ю. Морозов

Москва, 2022

Условие

Цель работы. Знакомство с технологией MPI. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

Математическая постановка:

$$\frac{d^2u(x,y,z)}{dx^2} + \frac{d^2u(x,y,z)}{dy^2} + \frac{d^2u(x,y,z)}{dz^2} = 0,$$

$u(x \le 0, y, z) = u_{left}$,

$u(x \ge l_x, y, z) = u_{right}$,

$u(x,\ y \le 0, z) = u_{front}$,

$u(x,\ y \ge l_y, z) = u_{back}$,

$u(x,\ y,\ z \le 0) = u_{down}$,

$u(x,\ y,\ z \ge l_z) = u_{up}$.

Над пространством строится регулярная сетка. С каждой ячейкой сопоставляется значение функции u в точке соответствующей центру ячейки. Граничные условия реализуются через виртуальные ячейки, которые окружают рассматриваемую область.

Поиск решения сводится к итерационному процессу:

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

где

$i = 1..n_x,\ j = 1..n_y, k = 1..n_z$,

$h_x = l_x n_x^{-1},\ h_y = l_y n_y^{-1},\ h_z = l_z n_z^{-1}$,

$u_{0,j,k}^{(n)} = u_{left}, u_{n_x+1,j,k}^{(n)} = u_{right}$,

$u_{i,0,k}^{(n)} = u_{front}, u_{i,n_y+1,k}^{(n)} = u_{back}$,

$u_{i,j,0}^{(n)} = u_{down}, u_{i,j,n_z+1}^{(n)} = u_{up}$,

$u_{i,j,k}^{(0)} = u^0$.

Процесс останавливается, как только

$$max_{i,j,k}\left|u_{i,j,k}^{(n+1)} - u_{i,j,k}^{(n)}\right| < \epsilon$$

Входные данные. На первой строке заданы три числа: размер сетки

процессов. Гарантируется, что при запуске программы количество процессов будет равно произведению этих трех чисел. На второй строке задается размер блока, который будет обрабатываться одним процессом: три числа. Далее задается путь к выходному файлу, в который необходимо записать конечный результат работы программы и точность ϵ. На последующих строках описывается задача: задаются размеры области l_x, l_y и l_z, граничные условия: u_{down}, u_{up}, u_{left}, u_{right}, u_{front} и u_{back}, и начальное значение u^0.

Выходные данные. В файл, определенный во входных данных, необходимо напечатать построчно значения ($u_{1,1,1}, u_{2,1,1}, \ldots, u_{1,2,1}, u_{2,2,1}, \ldots, u_{n_x-1,n_y,n_z}, u_{n_x,n_y,n_z}$) в ячейках сетки в формате с плавающей запятой с семью знаками мантиссы.

Пример:

Входной файл	mpi.out
1 1 1 3 3 3 mpi.out 1e-10 1.0 1.0 1.0 7.0 7.0 7.0 7.0 7.0 7.0 0.0	7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00 7.000000e+00
1 1 2 3 3 3 mpi.out 1e-10 1.0 1.0 2.0 7.0 0.0 5.0 0.0 3.0 0.0 5.0	4.627978e+00 4.194414e+00 3.239955e+00 4.728116e+00 4.177304e+00 3.095109e+00 3.795164e+00 3.214610e+00 2.407141e+00 3.845336e+00 3.121249e+00 2.150205e+00 3.768252e+00 2.831573e+00 1.746256e+00 2.828257e+00 1.908052e+00 1.133126e+00 3.554538e+00 2.705966e+00 1.793770e+00 3.376228e+00 2.268328e+00 1.267522e+00 2.498077e+00 1.440742e+00 7.373100e-01 3.399697e+00 2.497911e+00 1.638930e+00 3.168173e+00 1.987935e+00 1.059467e+00 2.343237e+00 1.232688e+00 5.824692e-01 3.177560e+00 2.254941e+00 1.482429e+00 2.901943e+00 1.701042e+00 8.799475e-01 2.160481e+00 1.041743e+00 4.653501e-01 2.508778e+00 1.670702e+00 1.120755e+00 2.204404e+00 1.139741e+00 5.713973e-01 1.675964e+00 6.908981e-01 2.879409e-01

Запись результатов в файл должна осуществляться одним процессом. Необходимо использовать последовательную пересылку данных по частям на пишущий процесс.
В программе допускается двойное использование памяти относительно размера блока обрабатываемого одним процессом.

Вариант 4.

Обмен граничными слоями через isend/irecv, контроль сходимости allgather;

Программное и аппаратное обеспечение

Компилятор nvcc версии 7.0(g++ версии 4.8.4) на 64-х битной Ubuntu 14.04 LTS.
Параметры графического процессора:
Compute capability : 6.1
Name : GeForce GTX 1050
Total Global Memory : 2096103424
Shared memory per block : 49152
Registers per block : 65536
Max threads per block : (1024, 1024, 64)
Max block : (2147483647, 65535, 65535)
Total constant memory : 65536
Multiprocessors count : 5

Метод решения

Для каждого процесса выделен свой участок памяти для обработки блока. У каждого процесса два равных по размеру блока, чтобы вычислить новые значения по старым, сохраненным во втором блоке.
Для получения граничных значений необходимо осуществлять общение между процессами.

Алгоритм решения:

1) Обмен граничными слоями между процессами.
2) Обновление значений внутри процессов.
3) Вычисление погрешности. И локально и по всей области с помощью обмена данными.

Описание программы

Макрос **CSC** - макрос для отслеживания ошибок со стороны GPU, вызывается около функций для cuda и выводит текст ошибки при cudaError_t не равным cudaSuccess.

В **main** вводом и выводом данных занимается один главный процесс **main_worker**. Основной алгоритм выполняется в цикле do while, заканчивающийся после погрешности, меньшей выбранного эпсилон.

edges_exchange - функция обмена граничными слоями между процессами (сохранение в буферы edge_buf_in и edge_buf_out). Ожидание конца получения данных в ней с помощью функции **recv_waiting**.

import_edges - функция перекачки текущих(старых) значений ячеек границ, полученных в **edges_exchange,** перед обновлением.

export_edges - функция передачи обновленных границ в нужный буфер для дальнейшей передачи соседям.

max_determine - функция подсчета разности до и после пересчета внутри одного процесса.

После пересчета значений используется удобная функция **MPI_Allgather**, которая вернет значение max_diff(полученных с помощью **max_determine**) по всем процессам. Далее поиск максимальной разницы (чтобы сошлись все процессы).

Результаты

Размер общей сетки тестов ниже не меняется.

Сетка / Расчет	Результат на MPI, мс	на CPU, мс
1 1 1 / 40 40 40	19861	9843
2 2 2 / 20 20 20	4943	9952
2 2 4 / 20 20 10	6224	9890

Код программы для CPU:

```
#include <iostream>
#include <string>
#include <fstream>
#include <algorithm>
#include <stdlib.h>
#include <iomanip>
#include <string.h>
#include <chrono>

using namespace std;
using namespace std::chrono;

const int ndims = 3;
const int ndims_x_2 = 6;

double u_next(double ux0, double ux1, double uy0, double uy1, double uz0, double
uz1, double h2x, double h2y, double h2z){
    double ans = (ux0 + ux1) * h2x;
    ans += (uy0 + uy1) * h2y;
    ans += (uz0 + uz1) * h2z;
    return ans;
}

double max_determine(double val1, double val2, double curr_max){
    double diff = val1 - val2;
    diff = diff < 0.0 ? -diff : diff;

    return diff > curr_max ? diff : curr_max;
}
```

```
void print_line(ostream& os, double* line, int size){
    for(int i = 0; i < size; ++i){
        os << line[i] << " ";
    }
}


int main(int argc, char **argv){
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int dimens[ndims], blocks[ndims];
    double l[ndims];
    double u[ndims_x_2];
    double u0, eps;
    string path;

    enum orientation{
        left = 0, right,
        front, back,
        down, up,
    };
    enum direction{
        dir_x = 0,
        dir_y,
        dir_z
    };

    cin >> dimens[dir_x] >> dimens[dir_y] >> dimens[dir_z];
    cin >> blocks[dir_x] >> blocks[dir_y] >> blocks[dir_z];
    cin >> path;
    cin >> eps;
    cin >> l[dir_x] >> l[dir_y] >> l[dir_z];
    cin >> u[down] >> u[up];
    cin >> u[left] >> u[right];
    cin >> u[front] >> u[back];
    cin >> u0;

    auto start = steady_clock::now();

    double max_diff = 0.0;

    int nx = dimens[dir_x]*blocks[dir_x];
    int ny = dimens[dir_y]*blocks[dir_y];
    int nz = dimens[dir_z]*blocks[dir_z];

    int sizex = nx + 2;
    int sizey = ny + 2;
    int sizez = nz + 2;

    double h2x, h2y, h2z;
    h2x = l[dir_x] / ((double)nx);
    h2y = l[dir_y] / ((double)ny);
    h2z = l[dir_z] / ((double)nz);

    h2x *= h2x;
    h2y *= h2y;
    h2z *= h2z;

    {
```

```
        double denuminator = 2.0*(1.0/h2x + 1.0/h2y + 1.0/h2z);
        h2x = 1.0 / (denuminator * h2x);
        h2y = 1.0 / (denuminator * h2y);
        h2z = 1.0 / (denuminator * h2z);
    }

    double* buffer0 = new double[sizex * sizey * sizez];
    double* buffer1 = new double[sizex * sizey * sizez];

    fill_n(buffer0, sizex * sizey * sizez, u0);

    int orr = 0;
    for(int i = 0; i < sizex; i += nx + 1, ++orr){
        for(int j = 1; j < ny + 1; ++j){
            for(int k = 1; k < nz + 1; ++k){
                buffer0[i + (j + k*sizey)*sizex] = u[orr];
                buffer1[i + (j + k*sizey)*sizex] = u[orr];
            }
        }
    }

    for(int j = 0; j < sizey; j += ny + 1, ++orr){
        for(int k = 1; k < nz + 1; ++k){
            for(int i = 1; i < nx + 1; ++i){
                buffer0[i + (j + k*sizey)*sizex] = u[orr];
                buffer1[i + (j + k*sizey)*sizex] = u[orr];
            }
        }
    }

    for(int k = 0; k < sizez; k += nz + 1, ++orr){
        for(int j = 1; j < ny + 1; ++j){
            for(int i = 1; i < nx + 1; ++i){
                buffer0[i + (j + k*sizey)*sizex] = u[orr];
                buffer1[i + (j + k*sizey)*sizex] = u[orr];
            }
        }
    }

    do{
        max_diff = 0.0;
        for(int k = 1; k <= nz; ++k){
            for(int j = 1; j <= ny; ++j){
                for(int i = 1; i <= nx; ++i){
                     buffer1[i + (j + k*sizey)*sizex] = u_next(buffer0[i - 1 + (j +
k*sizey)*sizex], buffer0[i + 1 + (j + k*sizey)*sizex], buffer0[i + (j - 1 +
k*sizey)*sizex], buffer0[i + (j + 1 + k*sizey)*sizex], buffer0[i + (j + k*sizey -
sizey)*sizex], buffer0[i + (j + k*sizey + sizey)*sizex], h2x, h2y, h2z);
                        max_diff = max_determine(buffer0[i + (j + k*sizey)*sizex],
buffer1[i + (j + k*sizey)*sizex], max_diff);
                }
            }
        }

        double* temp = buffer0;
        buffer0 = buffer1;
        buffer1 = temp;
    }while(max_diff >= eps);
```

```
    ofstream fout(path, ios::out);
    fout << std::scientific << std::setprecision(7);

    for(int k = 1; k <= nz; ++k){
        for(int j = 1; j <= ny; ++j){
            for(int i = 1; i <= nx; ++i){
                fout << buffer0[i + (j + k*sizey)*sizex] << " ";
            }
        }
    }
    fout << endl;
    delete[] buffer0;
    delete[] buffer1;
    fout.close();
    auto end = steady_clock::now();

    cout << "Inference time: ";
     cout << ((double)duration_cast<microseconds>(end - start).count()) / 1000.0 <<
"ms" << endl;
    return 0;
}
```

Выводы

1. Уменьшение времени выполнения программы при переходе на MPI на приведенных тестах незначительное. Нельзя сказать, что производительность увеличивается, т.к. обмен данными между процессами тяжелый и является существенным минусом.
2. Основная операция в MPI — это передача сообщений. В MPI реализованы практически все основные коммуникационные шаблоны: двухточечные (point-to-point), коллективные (collective) и односторонние (one-sided).
3. Префикс S - синхронный режим передачи данных. Операция передачи данных заканчивается только тогда, когда заканчивается прием данных. Функция нелокальная.
   Префикс B - буферизованный режим передачи данных. В адресном пространстве передающего процесса с помощью специальной функции создается буфер обмена, который используется в операциях обмена. Операция посылки заканчивается, когда данные помещены в этот буфер. Функция имеет локальный характер.
   Префикс R - согласованный или подготовленный режим передачи данных. Операция передачи данных начинается только тогда, когда принимающий процессор выставил признак готовности приема данных, инициировав операцию приема. Функция нелокальная.
   Префикс I - относится к неблокирующим операциям.
   Все функции передачи и приема сообщений могут использоваться в любой комбинации друг с другом.
4. Использование неблокирующих коммуникационных операций повышает безопасность с точки зрения возникновения тупиковых ситуаций, может увеличить скорость работы программы, совмещая выполнение вычислительных и коммуникационных операций. MPI_Isend, MPI_Irecv - неблокирующие.