Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Отчет**

по лабораторной работе №3

**«Построение минимальной выпуклой оболочки. Проход Грэхэма»**

**Выполнила:**

студентка группы 381808-2

Хисматулина К.Ф.

**Проверил:**

Нестеров А.Ю.

Нижний Новгород

2020

**Содержание**

[Постановка задачи 2](#_Toc531367741)

[Метод решения 3](#_Toc531367742)

[Схема распараллеливания 4](#_Toc531367743)

[Описание программной реализации 5](#_Toc531367744)

[Подтверждение корректности 6](#_Toc531367745)

[Результаты экспериментов 7](#_Toc531367746)

[Заключение 8](#_Toc531367747)

[Приложение 9](#_Toc531367748)

# Постановка задачи

В данной лабораторной работе необходимо реализовать построение минимальной выпуклой оболочки по алгоритму Грэхэма. Программа должна быть написана с возможностью запуска на нескольких процессах.

Задача построения выпуклой оболочки заключается в том, что необходимо получить некоторый вектор координат точек на плоскости и построить выпуклую область, которая включает все полученные точки и занимает минимальную возможную площадь.

Результат — упорядоченное множество вершин выпуклой оболочки.

Программа должна также уметь генерировать случайные множества точек и оценивать время выполнения.

Цель – получение навыков распараллеливания задач компьютерной графики средствами библиотеки MPI.

Критерии эффективности разработки:

* результаты тестирования корректности работы алгоритма (сравнение результатов работы линейной и параллельной версии программы)
* производительность параллельной версии программы выше, чем линейной(не достигнуто)

Требования:

* Должна использоваться библиотека MPI
* Схема распараллеливания не зависит от количества предоставленных ресурсов

# Метод решения

Если получено не более двух точек, выпуклая оболочка является точкой или отрезком и может быть построена сразу по полученным точкам. Иначе для нахождения выпуклой оболочки используется алгоритм Грэхэма.

Алгоритм Грэхэма:

1. Находим точку  нашего множества с самой маленькой у-координатой (если таких несколько, берем самую правую из них), добавляем в ответ.
2. Сортируем все остальные точки по полярному углу относительно  .
3. Добавляем в ответ  - самую первую из отсортированных точек.
4. Берем следующую по счету точку t. Пока t и две последних точки в текущей оболочке  и  образуют неправый поворот (вектора  t и ), удаляем из оболочки .
5. Добавляем в оболочку t.
6. Делаем п.5, пока не закончатся точки.

В программе выполняется вычисление косинусов углов, а не самих углов.

Косинус вычисляется как частное скалярного произведения векторов (значения вектора — разности координат точек) и произведения длин тех же векторов.

# Схема распараллеливания

В алгоритме Грэхэма для поиска точки выпуклой оболочки необходимо, чтобы были известны две предыдущие точки оболочки, потому между процессами постоянно должен осуществляться обмен данными.

В начале программы равномерно распределим векторы координат точек между всеми процессами.

Чтобы найти нижнюю левую точку p1, найдем на каждом процессе крайнюю левую точку из множества точек, которое известно процессу, соберем результаты на нулевом процессе и найдем в полученном множестве точек крайнюю левую точку снова, эта точка будет результатом поиска.

Чтобы, зная точки pi и pi+1 , найти точку pi+2 с минимальным углом между прямыми pi pi+1 иpi+1 pi+2 , найдем на каждом процессе такую точку из известного процессу множества точек, соберем результаты на нулевом процессе и повторим в полученном множестве поиск необходимой точки.

Когда мы завершаем построение выпуклой оболочки, необходимо оповестить все процессы об этом.

# Описание программной реализации

**Руководство пользователя**

Программа реализует линейную и параллельную версии алгоритма построение минимальной выпуклой оболочки.

Для начала работы с программой необходимо установить библиотеки MPI.

Запуск программы возможен только из интерфейса командной строки посредством введения в неё команды: [pathToMPI]/mpiexec.exe [–np [нужное кол-во процессов]] [путь к скомпилированному файлу].

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

# Подтверждение корректности

Для подтверждения корректности в программе предусмотрена автоматическая проверка полного соответствия результата работы линейной версии алгоритма результату работы параллельной версии алгоритма.

# Результаты экспериментов

|  |  |  |
| --- | --- | --- |
| Количество входных элементов | Последовательная реализация алгоритма | Параллельная реализация алгоритма |
| 10(4 процесса) | 0.000156 | 3.70999e-05 |
| 100(4 процесса) | 0.001458 | 0.000811 |
| 1000(4 процесса) | 0.105002 | 0.058932 |
| 10(8 процессов) | 0.000955 | 4.05e-05 |
| 100(8 процессов) | 0.001676 | 0.000668 |
| 1000(8 процессов) | 0.09285 | 0.05913 |

Из-за несовершенства параллельной части ускорение не достигается, однако при увеличении числа поступающих на вход элементов заметно уменьшается разница в скорости выполнения алгоритмов(до двух раз при 1000 элементах).

# Заключение

Мной был изучен и реализован алгоритм Грэхэма для поиска выпуклой оболочки.

Реализация программы эффективна только при больших объемах данных, потому что при работе программы на нескольких процессах возникают накладные расходы.

# Приложение

**Graham.h:**

#include <algorithm>

#include <vector>

#ifndef MODULES\_TASK\_3\_KHISMATULINA\_K\_GRAHAM\_GRAHAM\_H\_

#define MODULES\_TASK\_3\_KHISMATULINA\_K\_GRAHAM\_GRAHAM\_H\_

class Point {

public:

double x;

double y;

Point() { x = 0; y = 0; }

Point(double \_x, double \_y) : x(\_x), y(\_y) {}

Point& operator=(const Point& A) {

if (this != &A) {

x = A.x;

y = A.y;

}

return \*this;

}

bool operator==(const Point& p) {

return ((fabs(x - p.x) < 1e-7)) && (fabs(y - p.y) < 1e-7);

}

bool operator!=(const Point& p) {

return ((fabs(x - p.x) > 1e-7)) && (fabs(y - p.y) < 1e-7);

}

Point min(const Point& A, const Point& B) {

if (A.y < B.y) {

return A;

} else {

if ((fabs(A.y - B.y) < 1e-7) && (fabs(A.x - B.x) < 1e-7))

return A;

}

return B;

}

~Point() { x = 0; y = 0; }

};

class Vector {

public:

double X;

double Y;

Vector() { X = 0; Y = 0; }

Vector(Point A, Point B) { X = B.x - A.x; Y = B.y - A.y; }

Vector(double X\_, double Y\_) { X = X\_; Y = Y\_; }

~Vector() { X = Y = 0; }

};

double modul(const Vector& Vec);

double skalyar(const Vector& Vec1, const Vector& Vec2);

double cos(const Vector& A, const Vector& B);

bool cw(double x1, double y1, double x2, double y2, double x3, double y3);

std::vector<Point> GrahamPar(const std::vector<Point>& Points);

std::vector<Point> GrahamSeq(const std::vector<Point>& Points);

Point SearchMinPoint(const std::vector<Point>& Points);

std::vector<Point> Sort(const std::vector<Point>& Points);

std::vector<Point> GenPoints(int n);

#endif // MODULES\_TASK\_3\_KHISMATULINA\_K\_GRAHAM\_GRAHAM\_H\_

**Graham.cpp:**

// Copyright 2020 Khismatulina Karina

#include <mpi.h>

#include <vector>

#include <cmath>

#include <ctime>

#include <random>

#include "../../../modules/task\_3/khismatulina\_k\_graham/graham.h"

//#include <opencv2/opencv.hpp>

double modul(const Vector& Vec) {

return (sqrt(Vec.X \* Vec.X + Vec.Y \* Vec.Y));

}

double skalyar(const Vector& Vec1, const Vector& Vec2) {

return (Vec1.X \* Vec2.X + Vec1.Y \* Vec2.Y);

}

double cos(const Vector& A, const Vector& B) {

return (skalyar(A, B) / (modul(A) \* modul(B)));

}

Point SearchMinPoint(const std::vector<Point>& Points) {

Point min\_point = Points[0];

size\_t i = 0;

while (i < Points.size()) {

min\_point = min\_point.min(min\_point, Points[i]);

i += 1;

}

return min\_point;

}

std::vector<Point> GenPoints(int n) {

if (n < 1)

throw "slishkom malo!!!";

std::vector<Point> p(n);

std::mt19937 gen;

for (int i = 0; i < n; i++) {

p[i].x = gen();

p[i].y = gen();

}

return p;

}

std::vector<Point> Sort(const std::vector<Point>& Points) {

std::vector<Point> P = Points;

Point min\_point = SearchMinPoint(P);

std::vector<double> Cos(P.size());

Vector A(1, 0);

for (size\_t i = 0; i < P.size(); i++) {

Vector B(min\_point, Points[i]);

if (min\_point == Points[i]) {

Cos[i] = 2;

} else {

Cos[i] = cos(A, B);

}

}

for (size\_t i = 0; i < P.size() - 1; ++i) {

for (size\_t j = 0; j < P.size() - i - 1; ++j) {

if (Cos[j + 1] > Cos[j]) {

Point tmp = P[j + 1];

P[j + 1] = P[j];

P[j] = tmp;

double tmp\_cos = Cos[j + 1];

Cos[j + 1] = Cos[j];

Cos[j] = tmp\_cos;

}

}

}

for (size\_t i = 0; i < P.size() - 1; ++i) {

if (fabs(Cos[i + 1] - Cos[i]) < 0.000001) {

Vector Vec1(min\_point, P[i]);

Vector Vec2(min\_point, P[i + 1]);

if (modul(Vec1) > modul(Vec2)) {

Point temp = P[i + 1];

P[i + 1] = P[i];

P[i] = temp;

}

}

}

return P;

}

bool cw(double x1, double y1, double x2, double y2, double x3, double y3) {

return (x2 - x1) \* (y3 - y1) - (y2 - y1) \* (x3 - x1) < 0;

}

std::vector<Point> GrahamSeq(const std::vector<Point>& P) {

std::vector<Point> Points = Sort(P);

if (Points.size() < 2) {

return Points;

}

std::vector<double> Res;

Res.push\_back(Points[0].x);

Res.push\_back(Points[0].y);

Res.push\_back(Points[1].x);

Res.push\_back(Points[1].y);

if (Points.size() > 2) {

for (size\_t i = 2; i < Points.size(); ++i) {

while (cw(Res[Res.size() - 4], Res[Res.size() - 3], Res[Res.size() - 2],

Res[Res.size() - 1], Points[i].x, Points[i].y)) {

Res.pop\_back();

Res.pop\_back();

}

Res.push\_back(Points[i].x);

Res.push\_back(Points[i].y);

}

}

std::vector<Point> resPoint(Res.size() / 2);

for (size\_t i = 0; i < Res.size(); i += 2) {

resPoint[i / 2].x = Res[i];

resPoint[i / 2].y = Res[i + 1];

}

return resPoint;

}

std::vector<Point> GrahamPar(const std::vector<Point>& Points) {

std::vector<double> Res;

std::vector<Point> P = Sort(Points);

int points\_amount = P.size();

MPI\_Status status;

int rank, size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

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

int delta = points\_amount / size;

if (points\_amount >= size) {

std::vector<double> Vec;

for (size\_t i = 0; i < P.size(); ++i) {

Vec.push\_back(P[i].x);

Vec.push\_back(P[i].y);

}

std::vector<double> local\_vec(2 \* delta);

MPI\_Scatter(&Vec[0], 2 \* delta, MPI\_DOUBLE, &local\_vec[0], 2 \* delta, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

if (rank == (size - 1)) {

size\_t i = size \* delta;

while (i < P.size()) {

local\_vec.push\_back(P[i].x);

local\_vec.push\_back(P[i].y);

i += 1;

}

}

std::vector<Point> l\_P(local\_vec.size() / 2);

for (size\_t i = 0; i < local\_vec.size(); i += 2) {

l\_P[i / 2].x = local\_vec[i];

l\_P[i / 2].y = local\_vec[i + 1];

}

std::vector<Point> local\_res = GrahamSeq(l\_P);

local\_vec.clear();

for (size\_t i = 0; i < local\_res.size(); i++) {

local\_vec.push\_back(local\_res[i].x);

local\_vec.push\_back(local\_res[i].y);

}

int vec\_n = local\_vec.size();

std::vector<int> vec\_size(size);

int N = 0;

MPI\_Gather(&vec\_n, 1, MPI\_INT, &vec\_size[0], 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&vec\_n, &N, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

std::vector<double> global\_vec(N);

if (rank == 0) {

std::vector<int> address(size);

address[0] = 0;

for (int i = 1; i < size; i++) {

address[i] = vec\_size[i - 1];

}

for (int i = 2; i < size; i++) {

address[i] += address[i - 1];

}

for (int i = 0; i < vec\_n; i++) {

global\_vec[i] = local\_vec[i];

}

for (int i = 1; i < size; i++) {

MPI\_Recv(&global\_vec[address[i]], vec\_size[i], MPI\_DOUBLE, i, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

}

} else {

MPI\_Send(&local\_vec[0], vec\_n, MPI\_DOUBLE, 0, 1, MPI\_COMM\_WORLD);

}

if (rank == 0) {

std::vector<Point> global\_res(global\_vec.size() / 2);

size\_t i = 0;

while (i < global\_vec.size()) {

global\_res[i / 2].x = global\_vec[i];

global\_res[i / 2].y = global\_vec[i + 1];

i += 2;

}

std::vector<Point> res\_point = GrahamSeq(global\_res);

return res\_point;

}

} else {

std::vector<Point> res\_point = GrahamSeq(P);

return res\_point;

}

return P;

}

**Main.cpp:**

// Copyright 2020 Khismatulina Karina

#include <gtest-mpi-listener.hpp>

#include <gtest/gtest.h>

#include <math.h>

#include <stack>

#include <vector>

#include <iostream>

#include "../../../modules/task\_3/khismatulina\_k\_graham/graham.h"

//#include <opencv2/opencv.hpp>

TEST(khismatulina\_task\_3, test\_1\_error) {

int rank;

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

if (rank == 0) {

ASSERT\_ANY\_THROW(GenPoints(-1));

}

}

TEST(khismatulina\_task\_3, test\_2\_par\_10) {

int rank;

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

std::vector<Point> Points{ {0, 0}, {1, 5}, {3, 6}, {-1, -4}, {2, 2}, {1, 2.5}, {1, -1}, {0, 1}, {0, 3}, {4, -1} };

std::vector<Point> P = Sort(Points);

std::vector<Point> res = GrahamPar(P);

std::vector<Point> check{ {-1, 0}, {4, 5}, {3, 6}, {1, -4}, {0, 2} };

bool a = 1;

if (rank == 0) {

for (size\_t i = 0; i < res.size(); ++i)

if (res[i] != check[i])

a = 0;

EXPECT\_EQ(a, 1);

}

}

TEST(khismatulina\_task\_3, test\_3\_par\_100) {

int rank;

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

std::vector<Point> P = GenPoints(100);

std::vector<Point> res = GrahamPar(P);

std::vector<Point> check = GrahamSeq(P);

bool a = 1;

if (rank == 0) {

for (size\_t i = 0; i < res.size(); ++i)

if (res[i] != check[i])

a = 0;

EXPECT\_EQ(a, 1);

}

}

TEST(khismatulina\_task\_3, test\_4\_par\_200) {

int rank;

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

std::vector<Point> P = GenPoints(200);

std::vector<Point> res = GrahamPar(P);

std::vector<Point> check = GrahamSeq(P);

bool a = 1;

if (rank == 0) {

for (size\_t i = 0; i < res.size(); ++i)

if (res[i] != check[i])

a = 0;

EXPECT\_EQ(a, 1);

}

}

TEST(khismatulina\_task\_3, test\_5\_par\_500) {

int rank;

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

std::vector<Point> P = GenPoints(500);

std::vector<Point> res = GrahamPar(P);

std::vector<Point> check = GrahamSeq(P);

bool a = 1;

if (rank == 0) {

for (size\_t i = 0; i < res.size(); ++i)

if (res[i] != check[i])

a = 0;

EXPECT\_EQ(a, 1);

}

}

TEST(khismatulina\_task\_3, test\_for\_time) {

int rank;

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

std::vector<Point> P = GenPoints(1000);

double startTime, endTime;

startTime = MPI\_Wtime();

std::vector<Point> res = GrahamPar(P);

if (rank == 0) {

endTime = MPI\_Wtime();

std::cout << "par: " << endTime - startTime << std::endl;

}

if (rank == 0) {

bool a = 1;

startTime = MPI\_Wtime();

std::vector<Point> check = GrahamSeq(P);

endTime = MPI\_Wtime();

std::cout << "seq: " << endTime - startTime << std::endl;

EXPECT\_EQ(a, 1);

}

}

int main(int argc, char\*\* argv) {

::testing::InitGoogleTest(&argc, argv);

MPI\_Init(&argc, &argv);

::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);

::testing::TestEventListeners& listeners =

::testing::UnitTest::GetInstance()->listeners();

listeners.Release(listeners.default\_result\_printer());

listeners.Release(listeners.default\_xml\_generator());

listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);

return RUN\_ALL\_TESTS();

}