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**

**Отчет**

по лабораторной работе № 28

«Построение выпуклой оболочки – проход Грэхема»

**Выполнил:**

студент группы 381606-1

Воробьёв П.О.

**Проверил:**

Доцент кафедры МОСТ

Сысоев А.В.

Нижний Новгород

2018

Содержание

[Постановка задачи 3](#_Toc532921868)

[Метод решения 4](#_Toc532921869)

[Схема распараллеливания 5](#_Toc532921870)

[Описание программной реализации 6](#_Toc532921871)

[Подтверждение корректности 7](#_Toc532921872)

[Результаты экспериментов 8](#_Toc532921873)

[Заключение 9](#_Toc532921874)

[Приложение 10](#_Toc532921875)

# Постановка задачи

Разработать и реализовать программу для нахождения выпуклой оболочки N точек на плоскости.

Данная программа реализует следующий функционал:

* Ввод количества точек
* Границы прямоугольника, в пределах которых будут сгенерированы точки
* Сравнение параллельного и последовательного выполнения программы

# Метод решения

1. Для решения задачи построения выпуклой оболочки множества точек мы из точек a1…an выбираем точки с минимальной первой координатой, затем среди выбранных точек ищем точку с минимальной второй координатой. Таким образом, найденная точка, назовем её **c**, является лексикографическим минимумом точек a1…an .
2. Далее, начинаем обход границы против часовой стрелки, предварительно перенеся начало координат в точку **с.** После такого переноса на точках a1…an удаётся определить линейный порядок ( ≤ ), что для c1,c2 ∈{a1-c, … ,an–c} имеет место c1 ≤ c2, если либо det(c1,c2)>0, либо (det(c1,c2)=0)&( (c1,1)2+(c1,2)2) < (c2,1)2+(c2,2)2 ).
3. Как только линейный порядок на множестве точек определён, можно воспользоваться сортировкой, для того, чтобы упорядочить элементы по неубыванию в соответствии с введённым линейным порядком.
4. После этого мы произведем за время O(n) просмотр отсортированного массива с целью получения итоговых точек b1, …, bm исходя из того условия, что точка bi+1 располагается строго слева от вектора, идущего из точки bi-1 в точку bi, что эквивалентно требованию того, чтобы det(bi - bi-1, bi+1 - bi) > 0.

Стоит также заметить, что в алгоритме, на шаге 3, могут использоваться различные виды сортировки. В данной имплементации используется быстрая сортировка

# Схема распараллеливания

Для реализации параллельных вычислений множество точек рассылается равными частями всем процессам.

Далее, на каждом процессе происходит вычисление выпуклой оболочки подмножества точек. Затем, посчитанные результаты собираются на нулевом процессе и происходит вычисление выпуклой оболочки множества выпуклых оболочек подмножеств.

Вычисленный финальный результат является искомой выпуклой оболочкой множества N точек.

# Описание программной реализации

Для запуска программы необходимо ввести следующую команду в командную строку:

**mpiexec –n N program.exe** в директории, содержащей исполняемый файл.

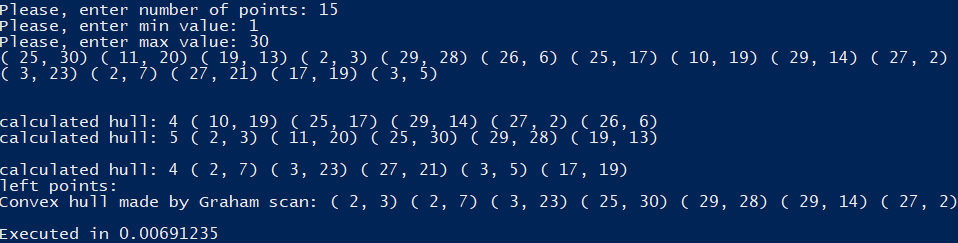
**N** означает количество процессов.

Либо **mpiexec –n N “{path}/program.exe”** где {path} – путь к исполняемому файлу.

После ввода данной команды, программа попросит пользователя ввести количество точек, минимальное и максимальное значение диапазона, в котором будут сгенерированы точки:



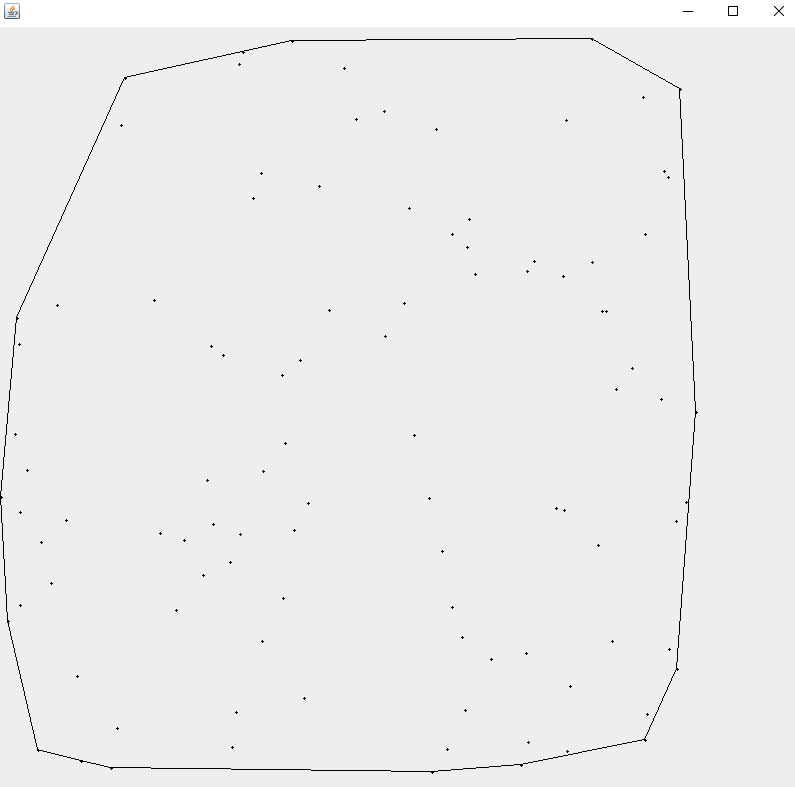
Затем программа выведет сгенерированное множество точек, размер вычисленной выпуклой оболочки подмножества на каждом процессе, а также саму выпуклую оболочку (первые N точек в массиве).



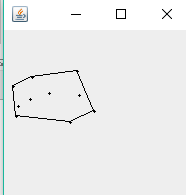
Далее, программа напечатает вычисленную итоговую выпуклую оболочку, а также время выполнения алгоритма.

# Подтверждение корректности

Для подтверждения корректности алгоритма была разработана графическая имплементация алгоритма на языке Java. (Вычисления производились последовательно. Т.к. для параллельных вычислений используется один и тот же метод вычисления выпуклой оболочки, можно утверждать, что результат выполнения на одном или нескольких процессах корректен).

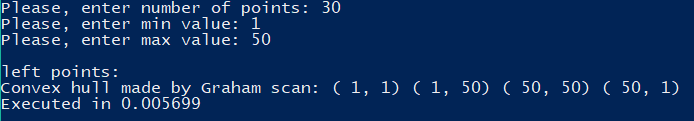


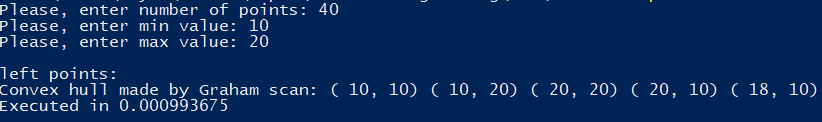
Результат работы алгоритма для 100 точек.



Для 10 точек.

Также было проведено испытание для набора точек с заранее известной выпуклой оболочкой. (Точки, входящие в границы квадрата, в пределах которого генерируются значения координат, а также 4 угловые точки)

В результате выполнения был выведен следующий набор точек:   




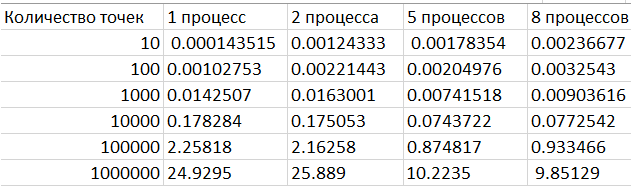
Исходя из полученного результата, можно сделать вывод что алгоритм работает корректно.

# 

# Результаты экспериментов

В качестве эксперимента был проведен замер работы времени алгоритма для различного числа процессов, а также для различного количества точек со значениями координат от 1 до 1 000 000.

В результате была построена следующая таблица:



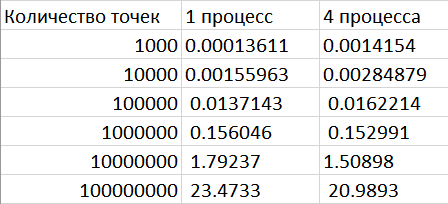
Из полученных данных можно сделать вывод, что результат сильно зависит от размера множества точек.   
Параллельный алгоритм показывает себя наиболее эффективно для большого количества точек, последовательное выполнение подходит для размера множества до 100 элементов.

Заметное ускорение параллельный алгоритм получает при количестве точек > 100 000.

Данный эксперимент проводился на случайно сгенерированном множестве точек, поэтому можно считать, что он не совсем корректен.

Повторный эксперимент был проведён на одинаковом множестве, для 1 и 4 процессов, версия Release, генерация точек в диапазоне от 1 до 100 000 000.

В результате была построена следующая таблица:



Исходя из полученных данных, можно сделать вывод о том, что алгоритм получает ускорение для множества точек размером > 1 000 000

# 

# Заключение

В ходе выполнения лабораторной работы, я изучил алгоритм Грехема для нахождения выпуклой оболочки множества точек, а также освоил способы распараллеливания этого алгоритма.

По результатам экспериментов видно, что параллельное исполнение программы приносит заметное ускорение при большом количестве точек.

# Приложение

*GrahamAlg.cpp*

#include "stdafx.h"

#include <iostream>

#include "mpi.h"

#include <vector>

#include <ctime>

using namespace std;

struct pt

{

int x;

int y;

};

class Point

{

private:

int x;

int y;

public:

int getY()

{

return y;

}

void setY(int y)

{

this->y = y;

}

int getX()

{

return x;

}

void setX(int x)

{

this->x = x;

}

Point(int x, int y)

{

this->setX(x);

this->setY(y);

}

Point()

{

}

static bool moreOrEqualThan(Point a, Point c, Point b)

{

return a.getX() \* b.getY() - a.getY() \* b.getX() + b.getX() \* c.getY() - b.getY() \* c.getX() + c.getX() \* a.getY() - c.getY() \* a.getX() <= 0;

}

static bool notEquals(Point a, Point b)

{

if (a.getX() != b.getX())

return true;

else if (a.getY() != b.getY())

return true;

else return false;

}

static int determinant(Point a, Point b, Point c)

{

return (b.getX() - a.getX()) \* (c.getY() - a.getY()) - (c.getX() - a.getX())\*(b.getY() - a.getY());

}

};

int partition(vector<Point> &a, int low, int high)

{

Point pivot = a[high];

int i = (low - 1);

for (int j = low; j < high; j++)

{

if (Point::moreOrEqualThan(pivot, a[j], a[0]))

{

i++;

Point tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

}

Point tmp = a[i + 1];

a[i + 1] = a[high];

a[high] = tmp;

return i + 1;

}

void quickSort(vector<Point> &a, int low, int high)

{

if (low < high)

{

int partIndex = partition(a, low, high);

quickSort(a, low, partIndex - 1);

quickSort(a, partIndex + 1, high);

}

}

int grahamScan(vector<Point>& a)

{

Point c = a[0];

int m = 0;

for (int i = 1; i < a.size(); i++)

{

if (a[i].getX() < c.getX())

{

c = a[i];

m = i;

}

else if (a[i].getX() == c.getX())

{

if (a[i].getY() < c.getY())

{

c = a[i];

m = i;

}

}

}

Point tmp = a[0];

a[0] = a[m];

a[m] = tmp;

m = 1;

quickSort(a, 0, a.size() - 1);

for (int i = 1; i < a.size(); i++)

{

if (Point::notEquals(a[i], a[m]))

{

if (m >= 1)

{

while (m >= 1 && Point::determinant(a[m - 1], a[m], a[i]) >= 0)

m = m - 1;

}

m = m + 1;

Point tmp = a[m];

a[m] = a[i];

a[i] = tmp;

}

}

return m + 1;

}

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

{

MPI\_Init(&argc, &argv);

int worldSize;

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

int rank;

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

int count = 2;

int arrOfBlocklengths[] = { 1, 1 };

MPI\_Aint arrOfDisps[] = { offsetof(pt, x), offsetof(pt, y) };

MPI\_Datatype arrOfTypes[] = { MPI\_INT, MPI\_INT };

MPI\_Datatype tmpType, pointType;

MPI\_Aint lb, extent;

MPI\_Type\_create\_struct(count, arrOfBlocklengths, arrOfDisps, arrOfTypes, &tmpType);

MPI\_Type\_get\_extent(tmpType, &lb, &extent);

MPI\_Type\_create\_resized(tmpType, lb, extent, &pointType);

MPI\_Type\_commit(&pointType);

double startTime = 0;

double endTime = 0;

if (rank == 0)

{

int numOfPoints = 0;

int min = 0;

int max = 0;

cout << "Please, enter number of points: ";

cin >> numOfPoints;

cout << "Please, enter min value: ";

cin >> min;

cout << "Please, enter max value: ";

cin >> max;

srand(time(nullptr));

vector<pt> data;

int sentPoints = 0;

startTime = MPI\_Wtime();

for (int i = 0; i < numOfPoints; i++)

{

pt p;

p.x = min + rand() % (max - min + 1);

p.y = min + rand() % (max - min + 1);

data.push\_back(p);

//cout << "( " << data[i].x << ", " << data[i].y << ") ";

}

cout << endl;

MPI\_Bcast(&numOfPoints, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

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

{

pt\* pointsToSend = new pt [data.size() / (worldSize - 1)];

//cout << "Sending points to " << i << " thread: ";

for (int j = 0; j < data.size() / (worldSize - 1); j++)

{

sentPoints++;

pt p = data[(i - 1) \*data.size()/(worldSize - 1) + j];

pointsToSend[j] = p;

//cout << "( " << p.x << ", " << p.y << ") ";

}

//cout << endl;

MPI\_Send(&pointsToSend[0], data.size() / (worldSize - 1), pointType, i, 0, MPI\_COMM\_WORLD);

}

//cout << "left points: ";

vector<Point> leftPoints;

vector<Point> receivedHulls;

for (int i = sentPoints; i < numOfPoints; i++)

{

Point p(data[i].x, data[i].y);

receivedHulls.push\_back(p);

//cout << "( " << p.getX() << ", " << p.getY() << ") ";

}

int newSize = 0;

if (worldSize > 1) {

newSize = data.size() / (worldSize - 1) + 1;

}

pt\* pointsToReceive = new pt[newSize];

for (int i = 0; i < worldSize - 1; i++)

{

MPI\_Recv(&pointsToReceive[0], newSize, pointType, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

int sizeOfHull = pointsToReceive[0].x;

for (int j = 1; j < sizeOfHull + 1; j++)

{

Point receivedPoint(pointsToReceive[j].x, pointsToReceive[j].y);

receivedHulls.push\_back(receivedPoint);

}

}

int hullSize = grahamScan(receivedHulls);

receivedHulls.resize(hullSize);

endTime = MPI\_Wtime();

cout << endl << "Convex hull made by Graham scan: ";

for (int i = 0; i < hullSize; i++)

{

//cout << "( " << receivedHulls[i].getX() << ", " << receivedHulls[i].getY() << ") ";

}

cout << endl << "Executed in " << endTime - startTime << endl;

}

else

{

int size = 0;

MPI\_Bcast(&size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

size = size / (worldSize - 1);

pt\* pts = new pt [size];

MPI\_Recv(&pts[0], size, pointType, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

vector<Point> points;

//cout << endl << "got points at " << rank << " ";

for (int i = 0; i < size; i++)

{

//cout << "( " << pts[i].x << ", " << pts[i].y << ") ";

Point point(pts[i].x, pts[i].y);

points.push\_back(point);

}

int newSize = grahamScan(points);

cout << endl << "calculated hull: " << newSize << " ";

pts = new pt[size + 1];

pts[0].x = newSize;

pts[0].y = 0;

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

{

pt p = { points[i - 1].getX(), points[i - 1].getY() };

pts[i] = p;

//cout << "( " << pts[i].x << ", " << pts[i].y << ") ";

}

cout << endl;

MPI\_Send(&pts[0], size + 1, pointType, 0, 0, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

}