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

**«Поиск выпуклой оболочки. Алгоритм Джарвиса»**

**Выполнил:**

студент группы 381606-2

Волков П.С.

**Проверил:**

Козинов Е.А.

Нижний Новгород

2018

**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

На координатной плоскости имеется набор точек вида (x, y). Необходимо найти наименьший выпуклый многоугольник, построенный на данном наборе точек и включающий в себя все остальные точки начального множества.

Основная задача состоит в реализации данной программы при помощи функций MPI. А также небольшого сравнения результатов тестов на различном количестве процессов и данных.

# Метод решения

Решение задачи проводится одним из хорошо известных алгоритмов для нахождения выпуклой оболочки. Алгоритмом Джарвиса. Рассмотрим шаги алгоритма подробно.

1. Найдем самую левую из всех нижних имеющихся точек. Понятно, что она входи в выпуклую оболочку. Назовем ее “Cur”.
2. Выберем некоторую мнимую точку на плоскости левее точки “Cur”. Назовем точку “Prev”.
3. Рассмотрим все точки, еще не вошедшие в оболочку. Каждую рассматриваемую точку будем называть “Next”.

3.1 Найдем косинус угла между точками “ Prev”, “Cur” и “Next”.

* 1. Если косинус угла больше, чем при просмотре предыдущей точки, то в качестве рассматриваемой на добавление в оболочку возьмем ее.
  2. Если нашлось несколько точек с максимальным значением косинуса, то выберем наиболее удаленную.
  3. После прохода по точкам добавим “Next” с максимальным косинусом угла в оболочку. Обозначим “Cur” именем “ Prev”. А “Next” обозначим именем “Cur”. Будем повторять пункт 3, пока не вернемся к изначально выбранной точке.

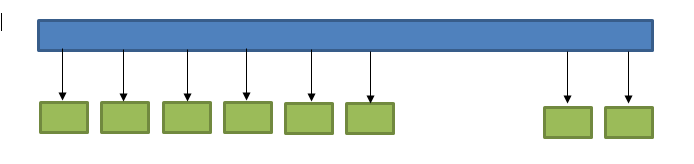
После подобного обхода точек формируется выпуклая оболочка. Обходя точки от первой к последней, мы будем проходить по точкам многоугольника выпуклой оболочки против часовой стрелки.

Суть параллельной версии программы будет состоять в том, что данные равномерно поделятся между процессами, что снизит нагрузку на каждый из них.

# Схема распараллеливания

Опишем алгоритм действия программы на N процессах.

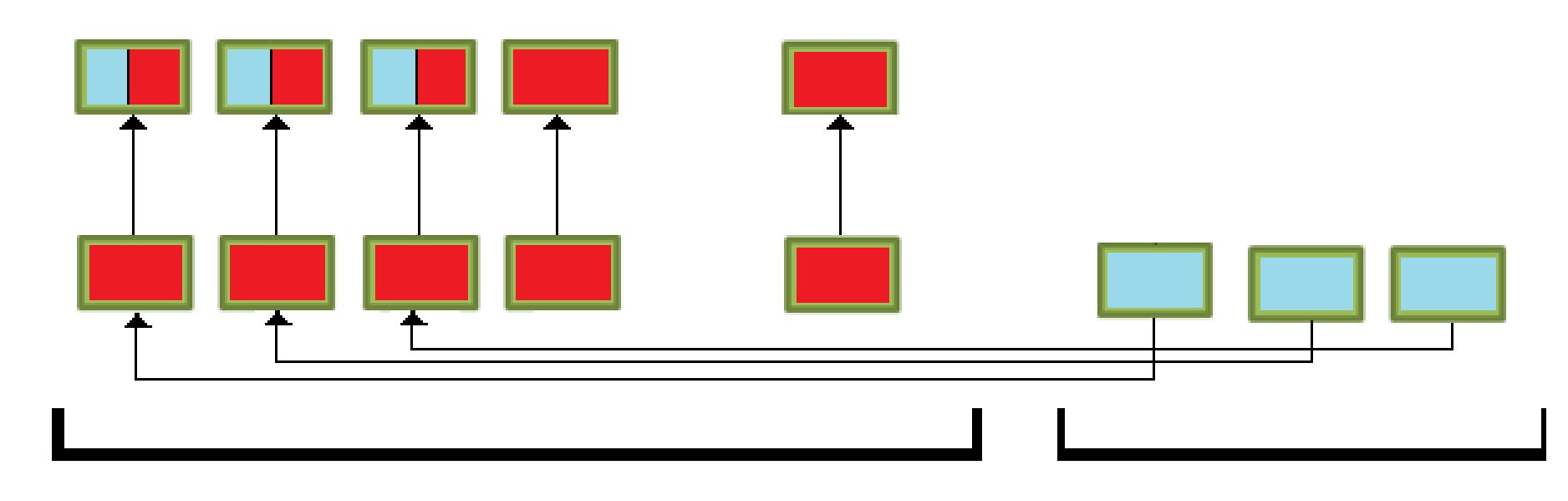
Начальный набор данных делится между всеми процессами.



После чего происходит применение алгоритма Джарвиса на каждом блоке данных.

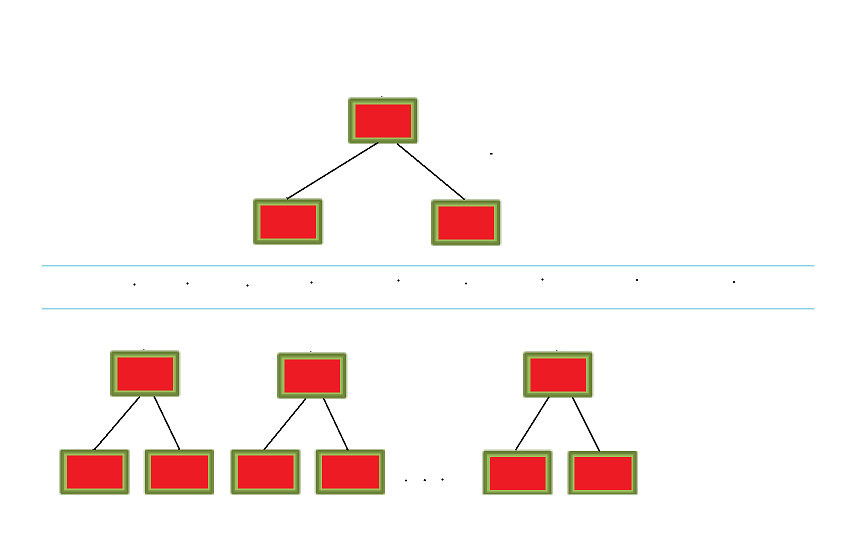
Далее будет происходить постепенное слияние данных из многих процессов в один. Это поможет не перегружать только какой-либо один процесс сразу всеми результатами вычислений остальных. Принято решение реализовать слияние данных следующим образом.

Выберем максимально возможное количество процессов, соответствующее степени двойки. Данные из остальных процессов перешлем первым.



**2i**Процессов **N – 2i**Процессов

Затем начнем попарно сливать данные всех процессов. После каждого слияния выполняется построения выпуклых оболочек на объединенных данных. Это позволяет распределить нагрузку между поделившимися данными процессами.



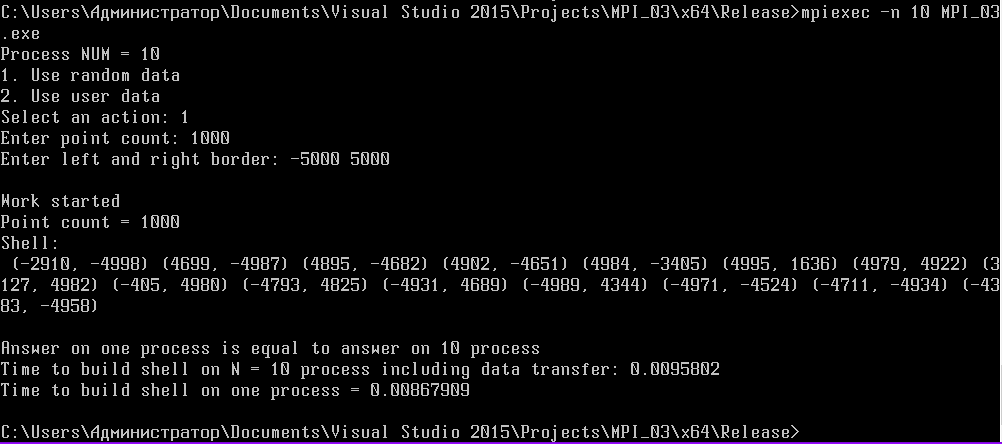
Т.к. количество оставшихся процессов соответствует степени двойки – построение подобного дерева слияния процессов возможно.

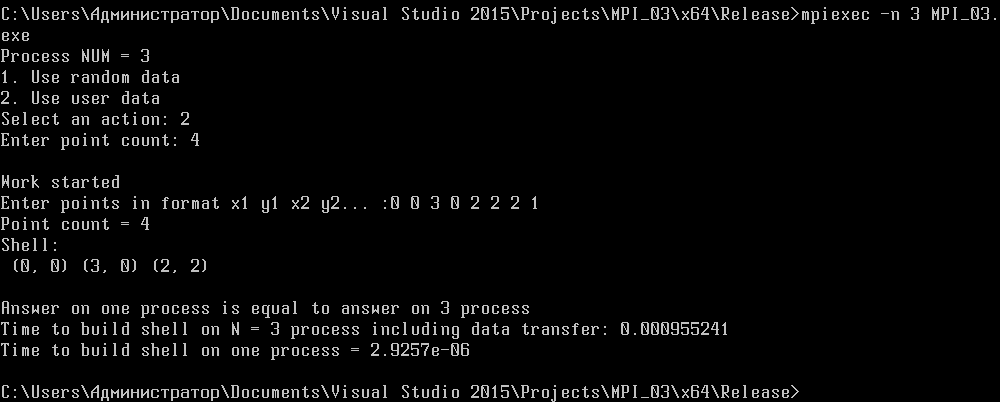
После последнего слияния и применения алгоритма Джарвиса – получим точки выпуклой оболочки.

# Описание программной реализации

**Руководство пользователя**

Для пользователя реализовано маленькое консольное меню. В меню можно выбрать ввод случайных данных или ввод данных пользователя.





**Руководство программиста**

Программу условно можно разделить на несколько блоков кода. Это функции и классы, написанные выше функции main. Они включат в себя функцию поиска выпуклой оболочки. Нахождения косинуса угла между двумя векторами и т.д. Строки (29-480).

Вторым условным блоком можно назвать отрисовку и взаимодействие с меню, а также ввод или генерацию данных. Строки (521-606).

Третьим блоком является многопроцессорное создание выпуклой оболочки по описанному алгоритму. Строки (606-957).

Последний блок – это проверка результата. Строки (957-1020).

В коде присутствует множество комментариев, что поможет разобраться в нем. Здесь же разберем лишь интерфейс функции построения оболочки.

int new\_shell(int \*buffer, int point\_count, vector<int> &out\_index)

Функция принимает на вход целочисленный буфер, размера point\_count \* 2. Где записаны данные о точках в формате X1 Y1 X2 Y2…

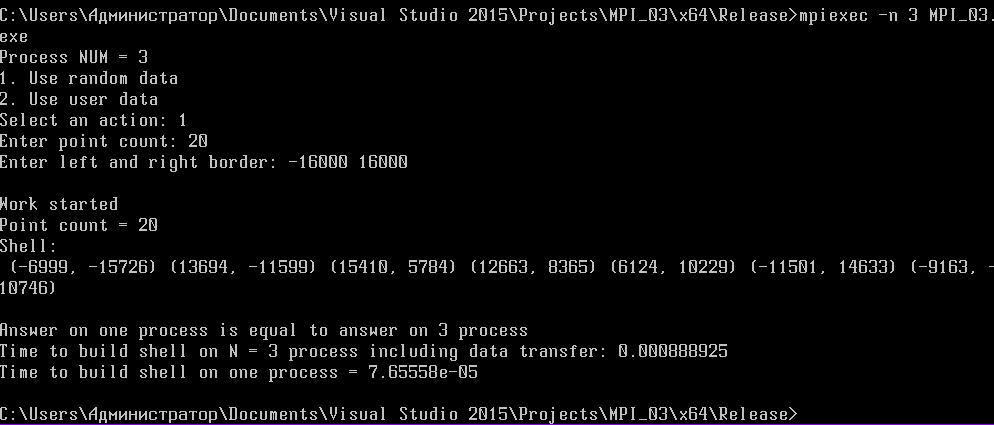
Функция возвращает количество элементов в выпуклой оболочке. Последним передается вектор целочисленных данных. В него запишутся индексы точек, входящих в выпуклую оболочку.

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

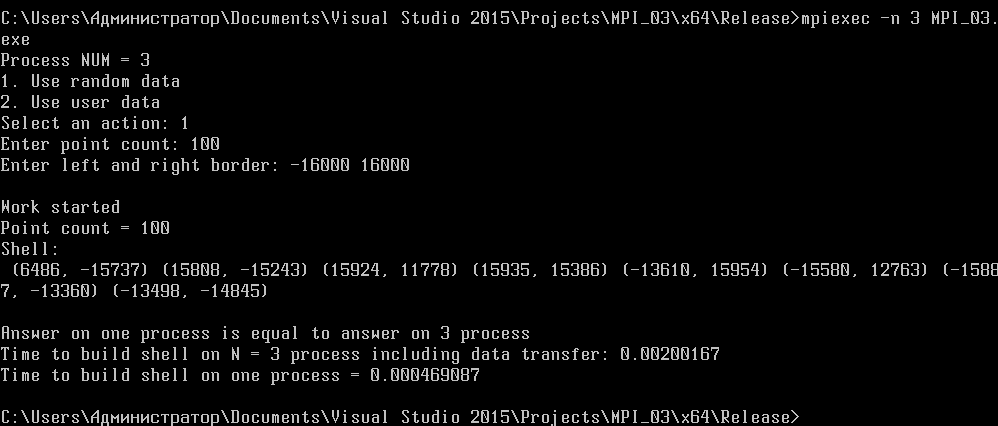
# Подтверждение корректности

Для подтверждения корректности в программе предусмотрено построение выпуклой оболочки лишь на одно процессе, без пересылки данных. После такого построение данные сравниваются. Т.к. алгоритм Джарвиса гарантирует нам, что выходные данные всегда будут представлять из себя набор точек в строго определенном порядке, то нам необходимо лишь сравнить полученные массивы данных линейно.

# Результаты экспериментов

По данным экспериментов видно, что на малых количествах данных не имеет значения количество задействованных процессов. Наоборот, при пересылке данных мы теряем большое количество времени.

Запуск поиска оболочки на 20-ти исходных и 3-х процессах.

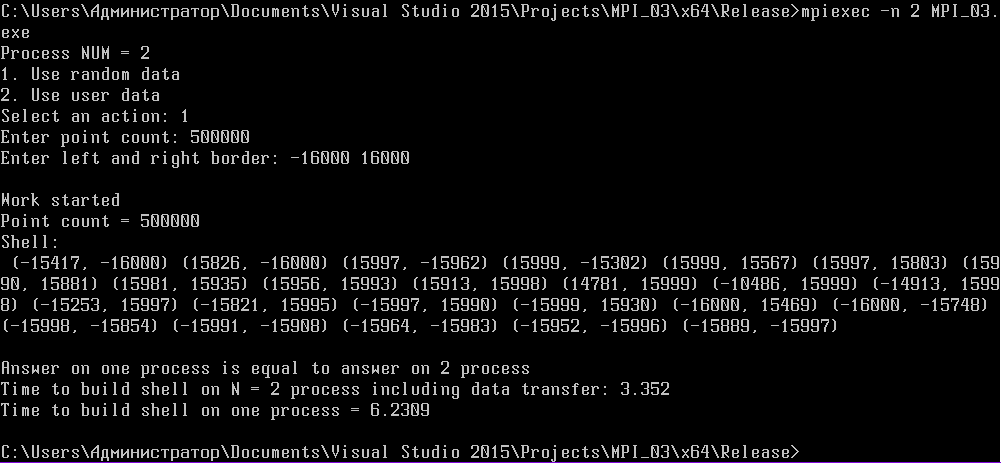


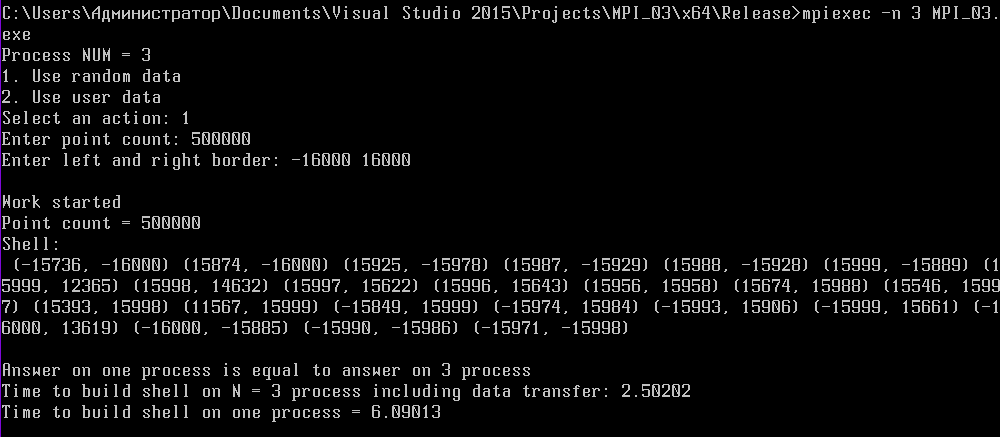
Запуск поиска оболочки на 100-не исходных и 3-х процессах.

На этих примерах видно, что на маленьких количествах точек параллельная программа показывает результаты на порядок хуже, чем программа, которая работает на одном процессе.

Переходным порогом является значение в примерно три тысячи элементов. После чего время выполнения параллельной версии уже не уступает по времени однопроцессорному поиску.

Обратим внимание на попытки использования программы с большим количеством данных.



Тут мы видим запуск поиска оболочки на 2х и 3х процессах. При примерно одинаковом времени построения оболочки на одном процессе в обоих случая, многопроцессное построение занимает различное время.

Время практически линейно пропорционально количеству использованных процессов.

К сожалению, на трех процессах ускорение заканчивается. Если пытаться использовать большее количество, то коэффициент ускорения будет стремиться к 3-м. Скорее всего, данный эффект вызван ограничением железа.

Теcты проводились на машине:

RAM: 4GB DDR3

CPU: AMD A6-3500 APU 2.1 GHz. 3 Ядра

OS: Windows 8.1

# Заключение

По итогам разработки программы можно сказать, что поставленная задача выполнена полностью. Создана параллельная версия поиска выпуклой оболочки алгоритмом Джарвиса.

Проведены и представлены самые значительны тесты на производительность.

# Приложение

#include <mpi.h>

#include <time.h>

#include <stdlib.h>

#include <iostream>

#include <string>

#include <vector>

#define N 10000

#define min\_size 1

#include <fstream>

using namespace std;

struct point;

void ConvexHullJarvis(const vector<point> &mas, vector<int> &convex\_hull, int n);

bool More(double x, double y);

bool Less(double x, double y);

bool Equal(double x, double y);

double dist(point, point);

double CosAngle(point a, point b, point c);

struct point

{

int x;

int y;

point(int \_x, int \_y)

{

x = \_x;

y = \_y;

}

point()

{

x = 0;

y = 0;

};

void operator=(point \_b)

{

x = \_b.x;

y = \_b.y;

}

};

bool operator!=(point a, point b)

{

if (a.x != b.x || a.y != b.y) return true;

return false;

}

bool operator==(point a, point b)

{

if (a.x == b.x && a.y == b.y) return true;

return false;

}

bool equal\_arrays(int \*a, int \*b, int n)

{

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

{

if (a[i] != b[i]) return false;

}

return true;

}

int shell(int \*buffer, int point\_count, vector<int> &out\_index)

{

vector<point> v\_p;

int i\_p1 = 0;

int i\_p2 = -1;

int i\_p3 = -1;

// Посчитать количество различных точек. Изи

point \*shell = new point[point\_count];

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

{

point point\_buffer;

point\_buffer.x = buffer[i \* 2];

point\_buffer.y = buffer[i \* 2 + 1];

v\_p.push\_back(point\_buffer);

if (v\_p[i] != v\_p[i\_p1])

{

if (i\_p2 == -1)

{

i\_p2 = i;

}

else

{

if (v\_p[i\_p2] != v\_p[i])

{

i\_p3 = i;

}

}

}

}

if (i\_p1 == 0 && i\_p2 == -1 && i\_p2 == -1)

{

out\_index.push\_back(0);

return 1;

}

if (i\_p1 == 0 && i\_p2 != -1 && i\_p3 == -1)

{

out\_index.push\_back(0);

out\_index.push\_back(i\_p2);

return 2;

}

ConvexHullJarvis(v\_p, out\_index, point\_count);

int shell\_count = out\_index.size() - 1;

return shell\_count;

}

int new\_shell(int \*buffer, int point\_count, vector<int> &out\_index)

{

out\_index.clear();

vector<point> points;

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

{

points.push\_back(point(buffer[i \* 2], buffer[i \* 2 + 1]));

}

// Получили массив точек.

// Начнем с проверки, что различных точек вообще больше трех

int p1 = 0, p2 = -1, p3 = -1;

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

{

if (p2 == -1 && points[i] != points[p1])

{

p2 = i;

continue;

}

if (p1 != -1 && p2 != 0 && points[i] != points[p1] && points[i] != points[p2])

{

p3 = i;

break;

}

}

if (p2 == -1)

{

out\_index.push\_back(0);

return 1;

}

if (p2 != -1 && p3 == -1)

{

// Только вот тут важен порядок, а то потом массивы не сойдутся

if (points[0].y < points[p2].y)

{

out\_index.push\_back(0);

out\_index.push\_back(p2);

return 2;

}

if (points[0].y == points[p2].y)

{

if (points[0].x < points[p2].x)

{

out\_index.push\_back(0);

out\_index.push\_back(p2);

return 2;

}

else

{

out\_index.push\_back(p2);

out\_index.push\_back(0);

return 2;

}

}

else

{

out\_index.push\_back(p2);

out\_index.push\_back(0);

return 2;

}

}

// Тут точно есть три различные точки

// Найдем самую леву и нижнюю из всех

int base = 0;

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

{

if (points[i].y < points[base].y)

{

base = i;

continue;

}

if (points[i].y == points[base].y && points[i].x < points[base].x)

{

base = i;

continue;

}

}

//Нашли самую левую из всех нижних

point cur, prev, next;

cur = points[base];

prev = cur; prev.x -= 1000;

out\_index.push\_back(base);

while (true)

{

bool find\_correct = false;

double max\_cos = -3;

double max\_dist;

int pretendent;

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

{

next = points[i];

// Проверить, нет ли ее еще в оболочке?

bool al\_exist = false;

for (int j = 0; j < out\_index.size(); j++)

{

if (points[i] == points[out\_index[j]])

{

al\_exist = true;

break;

}

}

if (al\_exist) continue; // Если наткнулись на точку в оболоче - пропускаем ее

//Если точка не в оболочке, то стоит проверять уже потихоньку угол

double cos = CosAngle(prev, cur, next);

// Если угол получился более развернутый

if (cos > max\_cos)

{

max\_cos = cos;

pretendent = i;

max\_dist = dist(cur, next);

continue;

}

// Если угол один и тот же

if (cos == max\_cos)

{

double pr\_d = dist(cur, next);

//Выбираем по максимальной дисстанции

if (pr\_d > max\_dist)

{

pretendent = i;

max\_dist = pr\_d;

}

continue;

}

}

// На всякий случай не забудем проверить угол с начальной, базовой точкой.

double base\_cos = CosAngle(prev, cur, points[base]);

if (base\_cos >= max\_cos)

{

//Все отлично, оболочка найдена

// Нужно прекратить поиск

break;

}

out\_index.push\_back(pretendent);

prev = cur;

cur = points[pretendent];

}

return int(out\_index.size());

}

void ConvexHullJarvis(const vector<point> &mas, vector<int> &convex\_hull, int n) // Старый вариант

{

// находим самую левую из самых нижних

int base = 0;

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

{

if (mas[i].y < mas[base].y)

base = i;

else

if (mas[i].y == mas[base].y &&

mas[i].x < mas[base].x)

base = i;

}

// эта точка точно входит в выпуклую оболочку

convex\_hull.push\_back(base);

point first;

first.x = mas[base].x;

first.y = mas[base].y;

point cur = first;

point prev = point(first.x - 1000, first.y);

do

{

double minCosAngle = -1e9; // чем больше угол, тем меньше его косинус

double maxLen = 1e9;

int next = -1;

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

{

double curCosAngle = CosAngle(prev, cur, mas[i]);

if (More(curCosAngle, minCosAngle))

{

next = i;

minCosAngle = curCosAngle;

maxLen = dist(cur, mas[i]);

}

else if (Equal(curCosAngle, minCosAngle))

{

double curLen = dist(cur, mas[i]);

if (More(curLen, maxLen))

{

next = i;

maxLen = curLen;

}

}

}

prev = cur;

cur = mas[next];

convex\_hull.push\_back(next);

} while (cur != first);

}

bool More(double x, double y)

{

if (x > y) return true;

return false;

}

bool Less(double x, double y)

{

if (x < y) return true;

else return false;

}

bool Equal(double x, double y)

{

if (x == y) return true;

else return false;

}

double dist(point a, point b)

{

return sqrt(double((a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y)));

}

int nod(int a, int b) // Наименьший общий делитель

{

a = abs(a);

b = abs(b);

int k = 0;

while (a != 0 && b != 0)

{

if (a > b) a %= b;

else b %= a;

}

k = a + b;

return k;

}

int one\_line(point a, point b, point c) //Лежат на одной линии

{

int ax = b.x - a.x;

int ay = b.y - a.y;

int bx = c.x - b.x;

int by = c.y - b.y;

// Нашли радиус-вектора.

int ka = nod(ax, ay);

int kb = nod(bx, by);

ax /= ka;

ay /= ka;

bx /= kb;

by /= kb;

if (ax == bx && ay == by) return 1; // Сонаправлены

if (ax == -bx && ay == -by) return -1; // противонаправлены

return 0;

}

double CosAngle(point a, point b, point c) // Косинус угла

{

if (a == b || b == c) return -1e10; // Бесконечность

int line = one\_line(a, b, c);

if (line == -1) return -1;

if (line == 1) return 1;

double ax, ay, bx, by;

ax = b.x - a.x;

ay = b.y - a.y;

bx = c.x - b.x;

by = c.y - b.y;

a.x \*= -1;

a.y \*= -1;

int k1, k2;

k1 = nod(ax, ay);

k2 = nod(bx, by);

ax /= k1; ay /= k1;

bx /= k2; by /= k2;

double cos;

cos = ((ax \* bx) + (ay \* by)) / (sqrt(ax \* ax + ay \* ay) \* sqrt(bx \* bx + by \* by));

return cos;

}

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

{

int taskid, numtasks;

int work\_size;

int \*point\_buffer = NULL;

int \*p\_work\_size = NULL;

int \*disp = NULL;

int \*double\_p\_work\_size = NULL;

MPI\_Init(&argc, &argv);

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

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

int all\_data\_count = 1000;

int a, b;

double start\_programm\_time;

double end\_programm\_time;

double cur\_proc\_time = 0;

double time\_to\_call\_shell = 0;

int act;

// Menu

if (taskid == 0)

{

cout << "Process NUM = " << numtasks << "\n";

cout << "1. Use random data\n";

cout << "2. Use user data\n";

//cout << "3. Use file data\n";

cout << "Select an action: ";

cin >> act;

if (act == 1)

{

cout << "Enter point count: ";

cin >> all\_data\_count;

cout << "Enter left and right border: ";

cin >> a;

cin >> b;

}

else

{

if (act == 2)

{

cout << "Enter point count: ";

cin >> all\_data\_count;

}

else

{

act == 1;

all\_data\_count = 1000;

a = -1000;

b = 1000;

}

}

cout << "\nWork started\n";

}

// Menu end

// Расчитаем объемы работ и разделение данных. Введем данные, если пользователь выбрал собственные точки

if (taskid == 0)

{

srand(time(NULL));

work\_size = all\_data\_count;

point\_buffer = new int[2 \* all\_data\_count];

if (act == 1)

{

for (int i = 0; i < all\_data\_count \* 2; i++)

{

int dif;

dif = b - a;

point\_buffer[i] = rand() % dif + a;

}

}

if (act == 2)

{

cout << "Enter points in format x1 y1 x2 y2... :";

for (int i = 0; i < all\_data\_count \* 2; i++)

{

cin >> point\_buffer[i];

}

}

cout << "Point count = " << all\_data\_count << "\n";

p\_work\_size = new int[numtasks];

int standart\_task;

standart\_task = all\_data\_count / numtasks;

// cout << "standart\_task = " << standart\_task << "\n";

int real\_proc\_count = 0;

if (standart\_task >= min\_size) // All is ok. Заданий достаточно

{

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

{

p\_work\_size[i] = standart\_task;

}

p\_work\_size[numtasks - 1] = all\_data\_count % numtasks + standart\_task;

real\_proc\_count = numtasks;

}

else // Точек недостаточно на все процессы

{

real\_proc\_count = all\_data\_count / min\_size;

if (all\_data\_count % min\_size != 0) real\_proc\_count++;

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

{

p\_work\_size[i] = min\_size;

}

if (all\_data\_count % min\_size != 0)

p\_work\_size[real\_proc\_count - 1] = all\_data\_count % min\_size;

else

{

p\_work\_size[real\_proc\_count - 1] = min\_size;

}

for (int i = real\_proc\_count; i < numtasks; i++)

{

p\_work\_size[i] = 0;

}

}

disp = new int[numtasks]; // Массив смещений для scatter\_v

disp[0] = 0;

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

{

disp[i] = disp[i - 1] + p\_work\_size[i - 1] \* 2;

}

double\_p\_work\_size = new int[numtasks];

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

{

double\_p\_work\_size[i] = p\_work\_size[i] \* 2;

}

start\_programm\_time = MPI\_Wtime();

}

MPI\_Scatter(p\_work\_size, 1, MPI\_INT, &work\_size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); //Режем и отправляем количество работы

int \*rec\_point\_buf = new int[work\_size \* 2];

MPI\_Scatterv(point\_buffer, double\_p\_work\_size, disp, MPI\_INT, rec\_point\_buf, work\_size \* 2, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Принимаем данные, с которыми будем работать

int \*shell\_buf = NULL;

int shell\_count;

vector<int> out\_index;

// Проводим первичные построения оболочек, если данных не ноль

if (work\_size != 0)

{

shell\_count = new\_shell(rec\_point\_buf, work\_size, out\_index);

shell\_buf = new int[shell\_count \* 2];

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

{

shell\_buf[i \* 2] = rec\_point\_buf[out\_index[i] \* 2];

shell\_buf[i \* 2 + 1] = rec\_point\_buf[out\_index[i] \* 2 + 1];

}

}

else

{

shell\_count = 0;

}

// Первичные оболочки построены. Теперь мы должны слить все данные, переодически перестраивая оболочки на новых данных.

// Вычислим в каждом, не лишний ли он?

// Т.е. сколько процессов попадают под степень двойки

int right\_num\_proc = 1;

while ( right\_num\_proc \* 2 <= numtasks )

{

right\_num\_proc \*= 2;

}

// В каждом процессе надо понять, будет ли он отдавать себя как хвост? Т.е. что он не входит в степень двойки

bool send\_tail = false;

if (taskid >= right\_num\_proc) send\_tail = true;

// Сколько же всего хвостов?

int tails\_count = numtasks - right\_num\_proc;

// А какие будут принимать хвосты?

bool rec\_tail = false;

if (taskid < tails\_count) rec\_tail = true;

// Так, осталось принять и отдать. Понять кому принять и кому отдать.

int send\_to, rec\_from;

int send\_count, rec\_count;

int \*union\_buffer = NULL;

if (send\_tail) // Отправка хвостов

{

send\_to = taskid - right\_num\_proc;

send\_count = shell\_count;

MPI\_Send(&send\_count, 1, MPI\_INT, send\_to, 8080, MPI\_COMM\_WORLD);

MPI\_Send(shell\_buf, shell\_count \* 2, MPI\_INT, send\_to, 8081, MPI\_COMM\_WORLD);

}

if (rec\_tail) // Прием хвостов

{

MPI\_Status st1; MPI\_Status st2;

rec\_from = right\_num\_proc + taskid;

MPI\_Recv(&rec\_count, 1, MPI\_INT, rec\_from, 8080, MPI\_COMM\_WORLD, &st1);

union\_buffer = new int[(rec\_count + shell\_count) \* 2];

MPI\_Recv(union\_buffer, rec\_count \* 2, MPI\_INT, rec\_from, 8081, MPI\_COMM\_WORLD, &st2);

for (int i = rec\_count; i < shell\_count + rec\_count; i++)

{

union\_buffer[i \* 2] = shell\_buf[i \* 2 - rec\_count \* 2];

union\_buffer[i \* 2 + 1] = shell\_buf[i \* 2 - rec\_count \* 2 + 1];

}

shell\_count = shell\_count + rec\_count;

}

// отлично, теперь бы надо все стандартизировать. Так что назовем все массивы одинаково

if ( !(rec\_tail || send\_tail)) // т.е. ничего не делали

{

union\_buffer = shell\_buf;

}

// Таак, теперь в каждом процессе есть shell\_count - количество элементов в оболочке и union\_buffer это оболочка. И все данные лежат в 2^n первых процессов

int level = 0;

bool live = true; // Не отдал ли процесс свои данные?

while ((1 << level) < right\_num\_proc) // Пока не дошли до определенной степени двойки

{

bool for\_send = false;

bool for\_rec = false;

int mask = 1 << level;

int k, k1;

if (live && taskid < right\_num\_proc && numtasks > 1)

{

// Как понять, принимаем мы или пересылаем данные? Хмм, ну, посомтрим бит номер level. Если 1, то шлем, если нет, то принимаем.

if (((mask & taskid) != 0) && live) for\_send = true;

if (((mask & taskid) == 0) && live) for\_rec = true;

if (for\_send)

{

live = false;

k1 = shell\_count;

k = shell\_count;

send\_to = 1 << level;

MPI\_Send(&k, 1, MPI\_INT, taskid - send\_to, 90, MPI\_COMM\_WORLD);

MPI\_Send(union\_buffer, k \* 2, MPI\_INT, taskid - send\_to, 91, MPI\_COMM\_WORLD);

}

if (for\_rec)

{

MPI\_Status st\_l;

rec\_from = 1 << level;

k = shell\_count;

MPI\_Recv(&k1, 1, MPI\_INT, taskid + rec\_from, 90, MPI\_COMM\_WORLD, &st\_l);

// cout << "id = " << taskid << " count = " << k1 + k << " \n";

int \*tmp\_name = new int[(k1 + k) \* 2];

MPI\_Recv(tmp\_name, k1 \* 2, MPI\_INT, taskid + rec\_from, 91, MPI\_COMM\_WORLD, &st\_l);

//int \*t = new int[0];

// Объединим текущий и пришедщий буферы

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

{

tmp\_name[i \* 2 + k1 \* 2] = union\_buffer[i \* 2];

tmp\_name[i \* 2 + k1 \* 2 + 1] = union\_buffer[i \* 2 + 1];

}

shell\_count = k + k1;

//Буфферы можно снова сократить

if (shell\_count > 0)

{

vector<int> index;

shell\_count = new\_shell(tmp\_name, shell\_count, index);

union\_buffer = new int[shell\_count \* 2];

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

{

union\_buffer[i \* 2] = tmp\_name[index[i] \* 2];

union\_buffer[i \* 2 + 1] = tmp\_name[index[i] \* 2 + 1];

}

// Заодно везде возвращаем нормально название

// Вот и привели все в порядок. Редуцировали точки

}

}

}

level += 1;

}

if (taskid == 0)

{

if (shell\_count > 0)

{

vector<int> index;

shell\_count = new\_shell(union\_buffer, shell\_count, index);

// Проводим последнее редуцирование, на случай, если процесс всего и был-то один

int \* tmp\_name = new int[shell\_count \* 2];

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

{

tmp\_name[i \* 2] = union\_buffer[index[i] \* 2];

tmp\_name[i \* 2 + 1] = union\_buffer[index[i] \* 2 + 1];

}

union\_buffer = tmp\_name;

}

end\_programm\_time = MPI\_Wtime();

// Конец работы парралельной части программы

// Вывод данных

string st;

st = "Shell:\n ";

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

{

st += "(" + to\_string(union\_buffer[i \* 2]) + ", " + to\_string(union\_buffer[i \* 2 + 1]) + ") ";

}

st += "\n\n";

cout << st;

}

// Check to equal on one process

if (taskid == 0)

{

vector<int> index\_n;

int one\_shell\_count;

double spent\_time\_one\_proc;

double str\_sh\_one\_pr;

double en\_sh\_one\_pr;

str\_sh\_one\_pr = MPI\_Wtime();

one\_shell\_count = new\_shell(point\_buffer, all\_data\_count, index\_n);

en\_sh\_one\_pr = MPI\_Wtime();

spent\_time\_one\_proc = en\_sh\_one\_pr - str\_sh\_one\_pr;

int \*one\_shell\_buffer;

one\_shell\_buffer = new int[one\_shell\_count \* 2];

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

{

one\_shell\_buffer[i \* 2] = point\_buffer[index\_n[i] \* 2];

one\_shell\_buffer[i \* 2 + 1] = point\_buffer[index\_n[i] \* 2 + 1];

}

if (one\_shell\_count == shell\_count)

{

if (equal\_arrays(one\_shell\_buffer, union\_buffer, one\_shell\_count))

{

cout << "Answer on one process is equal to answer on " << numtasks << " process\n";

}

else

{

cout << "Answers is not equal!!!\n";

}

}

else

{

cout << "Answers is not equal!!!\n";

}

cout << "Time to build shell on N = " << numtasks << " process including data transfer: " << end\_programm\_time - start\_programm\_time << "\n";

cout << "Time to build shell on one process = " << spent\_time\_one\_proc << "\n";

}

MPI\_Finalize();

return 0;

}