Федеральное государственное автономное образовательное учреждение высшего образования
"Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского" (ННГУ)

Институт информационных технологий, математики и механики

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«Построение выпуклой оболочки – проход Грэхема»

Выполнил: студент группы 381706-1
Силенко Дмитрий Игоревич
_____ Подпись

Проверил:
Доцент кафедры МОСТ, кандидат технических наук
_____ Сысоев А. В.

Нижний Новгород
2019.

Содержание

1. Введение

Основная цель данной работы – реализовать алгоритм прохода Грэхема, используемый для составления выпуклой оболочки из массива точек на плоскости.

Но для начала необходимо разобраться что же такое выпуклая оболочка.

Выпуклая оболочка множества точек — такое выпуклое множество точек, что все точки фигуры также лежат в нем. Необходимо отметить, что строить мы будем не просто выпуклую оболочку, а минимальную выпуклую оболочку.

Минимальная выпуклая оболочка множества точек — это минимальная по площади выпуклая оболочка.

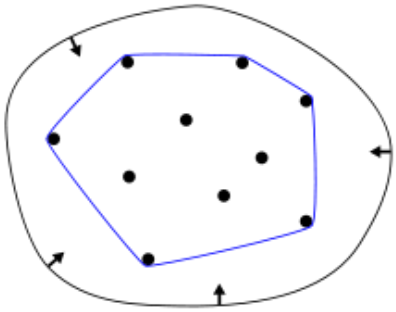

Рисунок 1 Выпуклая оболочка и минимальная выпуклая оболочка

По сути, минимальная выпуклая оболочка это знакомый всем со школы выпуклый многоугольник, состоящий из точек входного множества так, что все остальные точки этого множества лежат внутри него. Поскольку точки в множестве могут быть расположены как угодно, для успешного построения оболочки необходимо не только выбрать начальную вершину, но и отсортировать все остальные относительно нее. Проще всего это делать, оперируя полярными координатами заданных нам точек. Определим линейный порядок, относительно которого в дальнейшем будет производится сортировка. $c_1 \leq c_2$, если либо $\varphi_1 < \varphi_2$, либо $(\varphi_1 = \varphi_2) \& (r_1 \leq r_2)$.

2. Постановка задачи

Для точек a_1, …, a_n, где n≥1, a_i=($a_{i,1}$, $a_{i,2}$) ∈ R^2 при i=1, …, n, указать вершины b_1, …, b_m выпуклой оболочки Conv(a_1, …, a_n) в порядке их встречи при движении по ее границе. Заметим, что в общем случае Conv(a_1, …, a_n) будет многоугольником, а в вырожденных случаях может получиться отрезок или точка. В случае отрезка выходом решающего поставленную задачу алгоритма должны быть две являющиеся его концами точки, а в случае точки - сама эта точка.

Алгоритм построения Conv(a_1, …, a_n) необходимо распараллелить так, чтобы выпуклая оболочка строилась правильно для произвольного числа процессов, выполняющих ее построение.

3. Описание алгоритмов

3.1. Построение выпуклой оболочки с помощью прохода Грэхема

Для решения задачи построения Conv(a_1, …, a_n) мы из точек a_1, …, a_n выберем точки с минимальной первой координатой, среди которых затем найдем точку **c**, имеющую минимальную вторую координату. Таким образом, точка **c** = lexmin(a_1, … ,a_n) является лексикографическим минимумом точек a_1, … ,a_n и поэтому представляет собой вершину выпуклой оболочки Conv(a_1, …, a_n). Именно с нее мы и начнем обход границы против часовой стрелки, положив b_1=c и осуществив перед этим для удобства промежуточных вычислений параллельный перенос системы координат так, чтобы ее начало совпало с точкой **c**. После такого переноса на точках a_1, …, a_n удается определить такой линейный порядок ($\leq$), что для $c_1,c_2 \in \{a_1\text{-}c, \dots, a_n\text{–}c\}$ имеет место $c_1 \leq c_2$, если либо $\det(c_1,c_2)>0$, либо $(\det(c_1,c_2)=0)\&(\ (c_{1,1})^2+(c_{1,2})^2) < (c_{2,1})^2+(c_{2,2})^2\ )$.

Геометрический смысл такого упорядочения можно проиллюстрировать, если ввести полярную систему координат φ, r ($-\pi<\varphi\leq\pi$, $r\geq 0$) с центром в точке **c** и далее положить, что $c_1 \leq c_2$, если (φ_1,r_1) лексикографически не превосходит (φ_2,r_2), где числа φ_i, r_i являются полярными координатами точки c_i, i = 1,2.

Таким образом, $c_1 \leq c_2$, если либо $\varphi_1 < \varphi_2$, либо $(\varphi_1 = \varphi_2\ )\&(r_1 \leq r_2)$. Как только линейный порядок на элементах (точках) a_1, …, a_n определен, немедленно можем воспользоваться сортировкой, для того чтобы отсортировать эти элементы по не убыванию в соответствии с введенным линейным порядком. После этого мы произведем за время O(n) просмотр отсортированного массива с целью получения итоговых точек b_1, …, b_m исходя из того условия, что точка b_{i+1} располагается строго слева от вектора, идущего из точки b_{i-1} в точку b_i, что эквивалентно требованию того, чтобы $\det(b_i\text{-}b_{i-1},b_{i+1}\text{-}b_i)>0$. Рассмотрим этот процесс чуть поподробнее.

Создаем стек и заносим туда две первые точки из нашего множества. Для каждой следующей точки z:

1) Читаем y - верхушку стека и x – предпоследний элемент стека.
2) Если z находится слева от xy ($\det(x\text{-}y,z\text{-}x)>0$), то добавляем z в стек
3) Если z находится не слева от xy, то удалить y из стека и вернуться на пункт 1) (если в стеке было >= 3 элементов) или удалить y из стека y и добавить туда z (если в стеке было 2 элемента)

Как итог, все элементы, находящиеся в стеке и будут вершинами выпуклой оболочки. Остается лишь выполнить параллельный перенос системы координат обратно в (0;0).

3.2. Быстрая сортировка

Быстрая сортировка относится к алгоритмам «разделяй и властвуй». Алгоритм состоит из трёх шагов:

1. Выбрать элемент из массива. Назовём его опорным.
2. Разбиение: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

Для выбора опорного элемента и операции разбиения существуют разные подходы, влияющие на производительность алгоритма. Мы будем брать в качестве опорного элемента середину текущего подмассива.

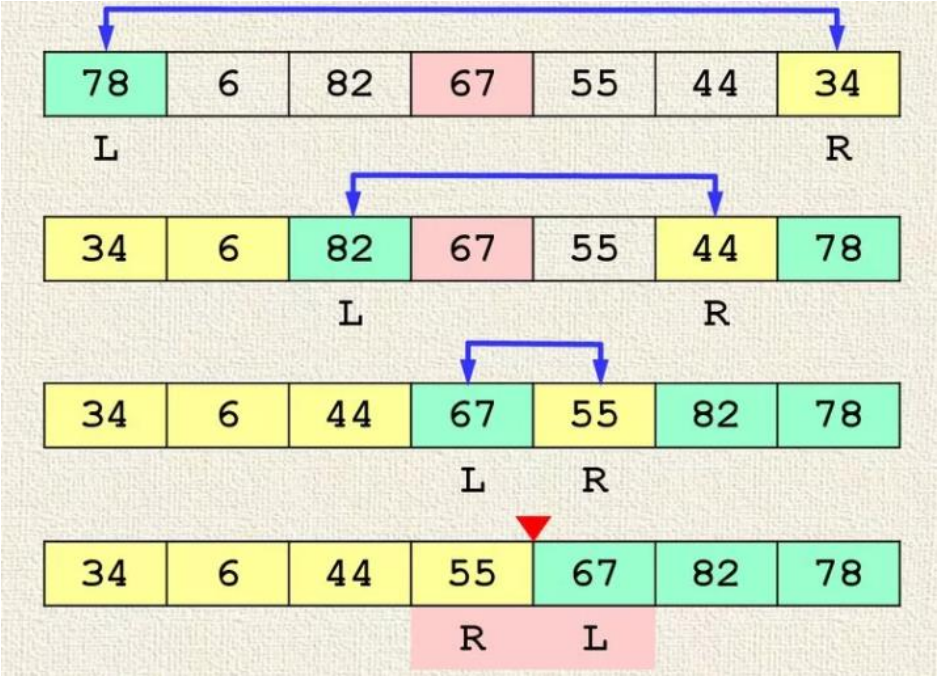

Рисунок 2 Быстрая сортировка, один проход

4. Схема распараллеливания

Организация параллельных вычислений происходит и при проведении подготовительных работ, и на моменте начала прохода Грэхема. Каждый процесс берет на себя некоторое количество точек (= все число точек / количество процессов), высчитывает их полярные координаты, после чего они сливаются в общий массив и сортируются. Далее процессы берут точки (такое же их количество, как и до этого) из уже отсортированных и на их основе строят свою выпуклую оболочку. Алгоритм здесь в точности повторяет последовательную версию. Затем отправляют полученный результат на нулевой процесс, который производит формирование уже конечного результата проходя по пришедшим ему точкам и сопоставляя их с текущим результатом

5. Описание программной реализации

Помимо основной функции, которая производит построение выпуклой оболочки в программе имеется еще ряд методов, которые так или иначе необходимы для корректной работы всего алгоритма. В первую очередь, это случайная генерация заданного числа точек (функция getRandomMas). Она нужна для проверки правильности работы всего алгоритма и используется в тестах. Две функции (SravnenieMore и SravnenieLess) по сути являются перегруженными операторами сравнения и необходимы для сортировки точек в лексикографическом порядке. Следующий метод (QuickSort) как раз и отвечает за быструю сортировку. И последней вспомогательной функцией является нахождение определителя (det) для трех точек, это используется при проверке, находится ли третья точка слева от прямой, образованной первыми двумя точками или нет. И наконец, основной метод построения выпуклой оболочки (ConvSort). Именно эта функция и распараллеливалась для работы на нескольких процессах.

6. Подтверждение корректности

Для подтверждения корректности работы программы были написаны 5 тестов с использованием библиотеки для модульного тестирования Google Testing Framework. 3 из них проверяют правильность работы при нестандартном размещении точек: когда они находятся на одной прямой, когда они размещены по сторонам одного прямоугольника и когда точка вообще одна. Еще один тест проверяет результат построение выпуклой оболочки при таком расположении точек, что ответ известен заранее. А последний генерирует случайные точки заданного количества и сравнивает последовательный алгоритм с параллельным. Все тесты проходятся успешно, что и является подтверждением корректности.

7. Эксперименты

Эксперименты проводились на ПК с следующими параметрами:

1. Операционная система: Windows 10 Домашняя
2. Процессор: Intel(R) Core™ i5-8250U CPU @ 1.60 GHz
3. Оперативная память: 4 Gb
4. Версия Visual Studio: 2017

В таблице 1 приведена зависимость времени работы алгоритма при разном числе процессов.

Количество элементов = 1 000 000.

Таблица 1. Время работы алгоритма в зависимости от числа процессов.

Количество процессов	Время работы последовательного алгоритма	Время работы параллельного алгоритма	Ускорение
1	0.307957	0.302119	1.019
2	0.303763	0.385126	0.7887
4	0.324456	0.661696	0.49
8	0.322574	1.11092	0.29

Исходя из этой таблицы можно сделать вывод, что программа работает не эффективно, причем, чем больше процессов выполняет программу, тем медленнее она работает. И это можно объяснить. Каждый процесс берет на себя какую-то часть точек и строит на их основе свою выпуклую оболочку, что несколько ускоряет работу конкретно этой части алгоритма. Но при этом, нулевой процесс должен сопоставить эти результаты с тем, что вычислил сам и исходя из этого достроить правильную выпуклую оболочку всего множества точек. По сути, нулевой процесс переделывает часть работы других процессов, чтобы получился правильный результат, а это отнимает больше времени, чем удается выиграть, деля изначальное множество точек на куски меньшего размера. И, соответственно, чем больше процессов, тем больше кусков необходимо сопоставлять нулевому процессу.

8. Заключение

Эта лабораторная работа позволила мне лучше разобраться в самом алгоритме построения выпуклой оболочки с математической точки зрения, а также понять, как правильно интерпретировать его в виде кода.

В данной работе мне удалось реализовать алгоритм построения выпуклой оболочки n точек на плоскости с использованием прохода Грэхема. А кроме этого, удалось распараллелить его для успешного построения выпуклой оболочки на любом количестве процессов. Быстрая сортировка используется из-за того, что в среднем она выдает хороший показатель скорости работы (O(n*log(n)), а также ее достаточно легко доработать для сравнения в лексикографическом порядке.

9. Литература

1. Груздев Д. В., Таланов В. А. Алгоритмы и структуры данных (лабораторные работы): [https://vk.com/doc51644906_515702336?hash=fc61c591076938632c&dl=0b9843bdc392de49ac], 2004.
2. Препарата Ф., Шеймос М. Вычислительная геометрия: Введение, 1989
3. Википедия: свободная электронная энциклопедия на русском языке: https://ru.wikipedia.org/wiki/Быстрая_сортировка

10. Приложение

10.1. Convex_Hull_Graham.h

```
// Copyright 2019 Silenko Dmitrii
#ifndef
MODULES_TASK_3_SILENKO_D_CONVEX_HULL_GRAHAM_CONVEX_HULL_GRAH
AM_H_
#define
MODULES_TASK_3_SILENKO_D_CONVEX_HULL_GRAHAM_CONVEX_HULL_GRAH
AM_H_

#include <stack>

double** getRandomMas(int count);

bool SravnenieMore(int num, double* _mid, double* fi, double* r);
bool SravnenieLess(int num, double* _mid, double* fi, double* r);

double** QuickSort(double** mas, int first, int last, double* fi, double* r);

double det(double* p1, double* p2, double* p3, int s);
std::stack<double*> ConvSort(double** mas, const int count);

#endif                                                                  //
MODULES_TASK_3_SILENKO_D_CONVEX_HULL_GRAHAM_CONVEX_HULL_GRAH
AM_H_
```

10.2. Convex_Hull_Graham.cpp

```
// Copyright 2019 Silenko Dmitrii

#include <mpi.h>
#include <iostream>
#include <random>
#include <cmath>
```

```
#include <stack>
#include <stdexcept>
#include "../../../modules/task_3/silenko_d_Convex_Hull_Graham/Convex_Hull_Graham.h"

double** getRandomMas(int count) {
  if (count <= 0)
    throw "Wrong count";
  double** mas = new double*[count];
  for (int i = 0; i < count; i++)
    mas[i] = new double[2];
  std::mt19937 seed;
  for (int i = 0; i < count; i++) {
    for (int j = 0; j < 2; j++) {
      mas[i][j] = seed() % 100;
    }
  }
  return mas;
}

bool SravnenieMore(int num, double* _mid, double* fi, double* r) {
  if (fi[num] > _mid[0]) {
    return true;
  } else if (fi[num] == _mid[0]) {
    if (r[num] > _mid[1])
      return true;
  }
  return false;
}

bool SravnenieLess(int num, double* _mid, double* fi, double* r) {
  if (fi[num] < _mid[0]) {
    return true;
  } else if (fi[num] == _mid[0]) {
    if (r[num] < _mid[1])
```

```
    return true;
  }
  return false;
}

double** QuickSort(double** mas, int first, int last, double* fi, double* r) {
  double* mid = new double[2];
  double tmp;
  int f = first, l = last;
  mid[0] = fi[(f + l) / 2];
  mid[1] = r[(f + l) / 2];
  do {
    while (SravnenieLess(f, mid, fi, r))
      f++;
    while (SravnenieMore(l, mid, fi, r))
      l--;

    if (f <= l) {
      tmp = r[f];
      r[f] = r[l];
      r[l] = tmp;
      tmp = fi[f];
      fi[f] = fi[l];
      fi[l] = tmp;

      tmp = mas[f][0];
      mas[f][0] = mas[l][0];
      mas[l][0] = tmp;
      tmp = mas[f][1];
      mas[f][1] = mas[l][1];
      mas[l][1] = tmp;

      f++;
      l--;
```

```
    }
  } while (f < l);
  if (first < l)
    QuickSort(mas, first, l, fi, r);
  if (f < last)
    QuickSort(mas, f, last, fi, r);
  return mas;
}

double det(double* p1, double* p2, double* p3, int size) {
  double diag1 = (p1[0] - p2[0]) * (p3[1] - p1[1]);
  double diag2 = (p3[0] - p1[0]) * (p1[1] - p2[1]);
  return (diag1 - diag2);
}

std::stack<double*> ConvSort(double** mas, const int count) {
  std::stack<double*> res;
  std::stack<double*> prom_res;
  if (count == 1) {
    res.push(mas[0]);
  } else if (count == 2) {
    res.push(mas[0]);
    res.push(mas[1]);
  } else {
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Status status;
    int delta;
    int ost;
    int f = 0;
    ost = count % size;

    if ((count < size) || (count / size < 3)) {
```

```
  ost = 0;
  f = -1;
  size = 1;
}

delta = count / size;

double* c = new double[2];
c[0] = mas[0][0];
c[1] = mas[0][1];
int m = 0;

for (int i = 1; i < count; i++) {
  if (mas[i][0] < c[0]) {
    c[0] = mas[i][0];
    c[1] = mas[i][1];
    m = i;
  } else {
    if (mas[i][0] == c[0]) {
      if (mas[i][1] < c[1]) {
        c[0] = mas[i][0];
        c[1] = mas[i][1];
        m = i;
      }
    }
  }
}

double* r = new double[count];
double* fi = new double[count];
double* tmp = mas[0];
mas[0] = mas[m];
mas[m] = tmp;
m = 0;
```

```
for (int i = 1; i < count; i++)
  for (int j = 0; j < 2; j++)
    mas[i][j] = mas[i][j] - c[j];

if (rank == 0) {
  for (int i = 1; i < delta + ost; i++) {
    r[i] = pow((mas[i][0] * mas[i][0]) + (mas[i][1] * mas[i][1]), 0.5);
    fi[i] = atan(mas[i][1] / mas[i][0]);
  }
} else {
  if (f == 0) {
    for (int i = delta*rank+ost; i < delta*rank + delta + ost; i++) {
      r[i] = pow((mas[i][0] * mas[i][0]) + (mas[i][1] * mas[i][1]), 0.5);
      fi[i] = atan(mas[i][1] / mas[i][0]);
    }
  }
}

if (size > 1) {
  MPI_Bcast(&r[1], delta + ost - 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&fi[1], delta + ost - 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

for (int i = 1; i < size; i++) {
  MPI_Bcast(&fi[delta * i + ost], delta, MPI_DOUBLE, i, MPI_COMM_WORLD);
  MPI_Bcast(&r[delta*i + ost], delta, MPI_DOUBLE, i, MPI_COMM_WORLD);
}

mas[0][0] = 0;
mas[0][1] = 0;
MPI_Barrier(MPI_COMM_WORLD);

QuickSort(mas, 1, count - 1, fi, r);
```

```
if (rank == 0) {
  prom_res.push(mas[0]);
  prom_res.push(mas[1]);
  for (int i = 2; i < delta + ost; i++) {
    double* last;
    double* beforelast;
    while (1) {
      last = prom_res.top();
      prom_res.pop();
      beforelast = prom_res.top();
      prom_res.push(last);

      double d = det(last, beforelast, mas[i], size);
      if (d > 0) {
        prom_res.push(mas[i]);
        break;
      } else {
        if (prom_res.size() >= 3) {
          prom_res.pop();
        } else {
          prom_res.pop();
          prom_res.push(mas[i]);
          break;
        }
      }
    }
  }
  if (f == 0) {
    int s;
    for (int i = 1; i < size; i++) {
      MPI_Recv(&s, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
      double* buf = new double[2 * s];
      MPI_Recv(&buf[0], 2 * s, MPI_DOUBLE, i, 9, MPI_COMM_WORLD, &status);
```

```
double* last;
double* beforelast;
int k = 1;
int flag = 0;
for (int j = 0; j < s; j++) {
  while (1) {
    last = prom_res.top();
    prom_res.pop();
    beforelast = prom_res.top();
    prom_res.push(last);
    double* tt = new double[2];
    tt[0] = buf[2 * s - (k + 1)];
    tt[1] = buf[2 * s - k];
    double d = det(last, beforelast, tt, size);
    if (d > 0) {
      flag++;
      prom_res.push(tt);
      break;
    } else {
      if (prom_res.size() >= 3) {
        if (flag > 0)
          flag--;
        prom_res.pop();
      } else {
        flag++;
        prom_res.pop();
        prom_res.push(tt);
        break;
      }
    }
    delete[] tt;
  }
  k += 2;
  if (flag >= 2)
```

```
        break;
      }
      for (int j = 2 * s - k; j > 0; j -= 2) {
       double* tt = new double[2];
       tt[0] = buf[j - 1];
       tt[1] = buf[j];
       prom_res.push(tt);
      }
     }
    }

    int s = prom_res.size();
    for (int i = 0; i < s; i++) {
     double* temp = prom_res.top();
     temp[0] += c[0];
     temp[1] += c[1];
     res.push(temp);
     prom_res.pop();
     temp = NULL;
    }
   } else {
    if (f == 0) {
     prom_res.push(mas[delta*rank + ost]);
     prom_res.push(mas[delta*rank + ost + 1]);
     for (int i = delta * rank + ost + 2; i < delta*rank + ost + delta; i++) {
      double* last;
      double* beforelast;
      while (1) {
       last = prom_res.top();
       prom_res.pop();
       beforelast = prom_res.top();
       prom_res.push(last);

       double d = det(last, beforelast, mas[i], 1);
```

```
          if (d > 0) {
            prom_res.push(mas[i]);
            break;
          } else {
            if (prom_res.size() >= 3) {
              prom_res.pop();
            } else {
              prom_res.pop();
              prom_res.push(mas[i]);
              break;
            }
          }
        }
      }

      int s = prom_res.size();
      MPI_Send(&s, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
      double* buf = new double[2 * prom_res.size()];
      for (int i = 0; i < 2 * s; i += 2) {
        tmp = prom_res.top();
        prom_res.pop();
        buf[i] = tmp[0];
        buf[i + 1] = tmp[1];
      }
      MPI_Send(&buf[0], 2 * s, MPI_DOUBLE, 0, 9, MPI_COMM_WORLD);
    }
   }
  }
  return res;
}
```

10.3. main.cpp

```
// Copyright 2019 Silenko Dmitrii

#include <gtest-mpi-listener.hpp>
```

```
#include <gtest/gtest.h>
#include <math.h>
#include <stack>
#include "../../../modules/task_3/silenko_d_Convex_Hull_Graham/Convex_Hull_Graham.h"

TEST(Convex_Hull_Graham_mpi, test_on_one_point) {
  double** mas = new double*[1];
  for (int i = 0; i < 1; i++)
    mas[i] = new double[2];
  mas[0][0] = 1;
  mas[0][1] = 1;

  std::stack<double*> res;
  std::stack<double*> result;
  double* tmp = new double[2];
  tmp[0] = 1;
  tmp[1] = 1;
  result.push(tmp);

  res = ConvSort(mas, 1);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    for (int i = 0; i < 1; i++) {
      tmp = res.top();
      double * tmp2 = result.top();
      for (int j = 0; j < 2; j++)
        EXPECT_EQ(tmp[j], tmp2[j]);
      res.pop();
      result.pop();
    }
  }
  delete[] tmp;
}
```

```
TEST(Convex_Hull_Graham_mpi, test_on_my_points) {
  double** mas = new double*[10];
  for (int i = 0; i < 10; i++)
    mas[i] = new double[2];
  mas[0][0] = -2;
  mas[0][1] = 1;
  mas[1][0] = -1;
  mas[1][1] = 3;
  mas[2][0] = -1;
  mas[2][1] = -3;
  mas[3][0] = 0;
  mas[3][1] = 0;
  mas[4][0] = 0;
  mas[4][1] = 4;
  mas[5][0] = 2;
  mas[5][1] = -2;
  mas[6][0] = 4;
  mas[6][1] = 4;
  mas[7][0] = 6;
  mas[7][1] = 3;
  mas[8][0] = 6;
  mas[8][1] = -3;
  mas[9][0] = 8;
  mas[9][1] = 1;

  std::stack<double*> res;
  std::stack<double*> result;
  double* tmp = new double[2];
  tmp[0] = -1;
  tmp[1] = 3;
  result.push(tmp);
  double* tmp0 = new double[2];
  tmp0[0] = 0;
```

```
tmp0[1] = 4;
result.push(tmp0);
double* tmp9 = new double[2];
tmp9[0] = 4;
tmp9[1] = 4;
result.push(tmp9);
double* tmp8 = new double[2];
tmp8[0] = 6;
tmp8[1] = 3;
result.push(tmp8);
double* tmp7 = new double[2];
tmp7[0] = 8;
tmp7[1] = 1;
result.push(tmp7);
double* tmp6 = new double[2];
tmp6[0] = 6;
tmp6[1] = -3;
result.push(tmp6);
double* tmp5 = new double[2];
tmp5[0] = -1;
tmp5[1] = -3;
result.push(tmp5);
double* tmp4 = new double[2];
tmp4[0] = -2;
tmp4[1] = 1;
result.push(tmp4);

res = ConvSort(mas, 10);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  int s = result.size();
  for (int i = 0; i < s; i++) {
```

```
      tmp = res.top();
      double* tmp2 = result.top();
      for (int j = 0; j < 2; j++)
        EXPECT_EQ(tmp[j], tmp2[j]);
      res.pop();
      result.pop();
    }
  }
  delete[] tmp;
  delete[] tmp8;
  delete[] tmp0;
  delete[] tmp9;
  delete[] tmp7;
  delete[] tmp6;
  delete[] tmp5;
  delete[] tmp4;
}

TEST(Convex_Hull_Graham_mpi, test_on_my_points_1) {
  double** mas = new double*[10];
  for (int i = 0; i < 10; i++)
    mas[i] = new double[2];
  mas[0][0] = -3;
  mas[0][1] = 3;
  mas[1][0] = 0;
  mas[1][1] = 3;
  mas[2][0] = 0;
  mas[2][1] = -1;
  mas[3][0] = -3;
  mas[3][1] = 1;
  mas[4][0] = 4;
  mas[4][1] = 3;
  mas[5][0] = 5;
  mas[5][1] = -1;
```

```
mas[6][0] = -3;
mas[6][1] = -1;
mas[7][0] = 2;
mas[7][1] = 3;
mas[8][0] = 5;
mas[8][1] = 2;
mas[9][0] = 2;
mas[9][1] = -1;

std::stack<double*> res;
std::stack<double*> result;
double* tmp = new double[2];
tmp[0] = -3;
tmp[1] = 3;
result.push(tmp);
double* tmp0 = new double[2];
tmp0[0] = 4;
tmp0[1] = 3;
result.push(tmp0);
double* tmp9 = new double[2];
tmp9[0] = 5;
tmp9[1] = 2;
result.push(tmp9);
double* tmp8 = new double[2];
tmp8[0] = 5;
tmp8[1] = -1;
result.push(tmp8);
double* tmp7 = new double[2];
tmp7[0] = -3;
tmp7[1] = -1;
result.push(tmp7);

res = ConvSort(mas, 10);
int rank;
```

```
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    int s = result.size();
    for (int i = 0; i < s; i++) {
      tmp = res.top();
      double* tmp2 = result.top();
      for (int j = 0; j < 2; j++)
        EXPECT_EQ(tmp[j], tmp2[j]);
      res.pop();
      result.pop();
    }
  }
  delete[] tmp;
  delete[] tmp8;
  delete[] tmp0;
  delete[] tmp9;
  delete[] tmp7;
}

TEST(Convex_Hull_Graham_mpi, test_on_one_line) {
  double** mas = new double*[12];
  for (int i = 0; i < 12; i++)
    mas[i] = new double[2];
  mas[0][0] = -1;
  mas[0][1] = 0;
  mas[1][0] = 3;
  mas[1][1] = 0;
  mas[2][0] = 5.5;
  mas[2][1] = 0;
  mas[3][0] = -2.5;
  mas[3][1] = 0;
  mas[4][0] = -7;
  mas[4][1] = 0;
```

```
mas[5][0] = 19;
mas[5][1] = 0;
mas[6][0] = 1;
mas[6][1] = 0;
mas[7][0] = -25;
mas[7][1] = 0;
mas[8][0] = 9;
mas[8][1] = 0;
mas[9][0] = 25;
mas[9][1] = 0;
mas[10][0] = 6;
mas[10][1] = 0;
mas[11][0] = 17;
mas[11][1] = 0;

std::stack<double*> res;
std::stack<double*> result;
double* tmp = new double[2];
tmp[0] = 25;
tmp[1] = 0;
result.push(tmp);
double* tmp0 = new double[2];
tmp0[0] = -25;
tmp0[1] = 0;
result.push(tmp0);

res = ConvSort(mas, 12);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  int s = result.size();
  for (int i = 0; i < s; i++) {
    tmp = res.top();
    double * tmp2 = result.top();
```

```
      for (int j = 0; j < 2; j++)
        EXPECT_EQ(tmp[j], tmp2[j]);
      res.pop();
      result.pop();
    }
  }
  delete[] tmp;
  delete[] tmp0;
}

TEST(Convex_Hull_Graham_mpi, test_on_my_random_points) {
  int count = 1000000;
  double** mas = new double*[count];
  for (int i = 0; i < count; i++)
    mas[i] = new double[2];
  mas = getRandomMas(count);
  double** newmas = new double*[count];
  for (int i = 0; i < count; i++) {
    newmas[i] = new double[2];
    for (int j = 0; j < 2; j++)
      newmas[i][j] = mas[i][j];
  }

  std::stack<double*> res;
  std::stack<double*> tmp_result;
  std::stack<double*> result;
  double start_solo = 0;
  double end_solo = 0;
  double* tmp = NULL;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) {
```

```
start_solo = MPI_Wtime();
double* c = new double[2];
c[0] = mas[0][0];
c[1] = mas[0][1];
int m = 0;

for (int i = 1; i < count; i++) {
  if (mas[i][0] < c[0]) {
    c[0] = mas[i][0];
    c[1] = mas[i][1];
    m = i;
  } else {
    if (mas[i][0] == c[0]) {
      if (mas[i][1] < c[1]) {
        c[0] = mas[i][0];
        c[1] = mas[i][1];
        m = i;
      }
    }
  }
}
tmp = mas[0];
mas[0] = mas[m];
mas[m] = tmp;
m = 0;
for (int i = 1; i < count; i++)
  for (int j = 0; j < 2; j++)
    mas[i][j] = mas[i][j] - c[j];
mas[0][0] = 0;
mas[0][1] = 0;

double* r = new double[count];
double* fi = new double[count];
r[0] = 0;
```

```
fi[0] = 0;
for (int i = 1; i < count; i++) {
  r[i] = pow((mas[i][0] * mas[i][0]) + (mas[i][1] * mas[i][1]), 0.5);
  fi[i] = atan(mas[i][1] / mas[i][0]);
}

QuickSort(mas, 1, count - 1, fi, r);

tmp_result.push(mas[0]);
tmp_result.push(mas[1]);
for (int i = 2; i < count; i++) {
  double* last;
  double* beforelast;
  while (1) {
    last = tmp_result.top();
    tmp_result.pop();
    beforelast = tmp_result.top();
    tmp_result.push(last);

    double d = det(last, beforelast, mas[i], 1);
    if (d > 0) {
      tmp_result.push(mas[i]);
      break;
    } else {
      if (tmp_result.size() >= 3) {
        tmp_result.pop();
      } else {
        tmp_result.pop();
        tmp_result.push(mas[i]);
        break;
      }
    }
  }
}
```

```
  int s = tmp_result.size();
  for (int i = 0; i < s; i++) {
    double* temp = tmp_result.top();
    temp[0] += c[0];
    temp[1] += c[1];
    result.push(temp);
    tmp_result.pop();
  }
  end_solo = MPI_Wtime();
 }

 MPI_Barrier(MPI_COMM_WORLD);

 double start = MPI_Wtime();
 res = ConvSort(newmas, count);
 double end = MPI_Wtime();

 if (rank == 0) {
  std::cout << "Runtime solo = " << end_solo - start_solo << "\n";
  std::cout << "Runtime multi = " << end - start << "\n";
  int s = result.size();
  for (int i = 0; i < s; i++) {
    tmp = res.top();
    double * tmp2 = result.top();
    for (int j = 0; j < 2; j++)
      EXPECT_EQ(tmp[j], tmp2[j]);
    res.pop();
    result.pop();
  }
 }
 delete[] tmp;
}
```

```
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();
}
```