МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)

Институт информационных технологий, математики и механики

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«Построение выпуклой оболочки для компонент бинарного изображения»

Выполнил:

Студент 3 курса, группы 3821Б1ФИ3:

Володин Евгений Владимирович

Проверил:

Доцент кафедры высокопроизводительных вычислений и системного программирования, кандидат технических наук:

Сысоев А.В.

Нижний Новгород

2023

Содержание

1 Введение

В данной лабораторной работе рассматривается задача построения выпуклой оболочки для компонент бинарного изображения. Эта задача относится к области компьютерного зрения, которая занимается анализом изображений и видео с помощью компьютерных алгоритмов. Компьютерное зрение имеет широкое применение в различных сферах, таких как медицина, робототехника, промышленность и безопасность.

Прежде чем перейти к самой лабораторной работе, стоит ввести термины, определяющие её суть.

Бинарное изображение — это изображение, состоящее из двух цветов, обычно черного и белого. Любое бинарное изображение состоит из компонент, количество которых может быть различно.

Компонента бинарного изображения — это связная область одного цвета, например, белая фигура на черном фоне. Чтобы мы могли работать с конкретной компонентой бинарного изображения, их (компоненты) необходимо отличать друг от друга. За это отвечает маркировка связных компонент бинарного изображения, т.е. процесс присвоения уникальных номеров каждой компоненте.

Выпуклая оболочка множества точек на плоскости — это наименьшее выпуклое множество, содержащее все эти точки. Построение выпуклой оболочки для компонент бинарного изображения позволяет выделить их форму и границы, а также получить полезную информацию о их размере, ориентации, количестве и т.д.

2 Постановка задачи

Целью данной работы является исследование алгоритма построения минимальной выпуклой оболочки для компонент бинарного изображения с использованием последовательного и параллельного подходов, а также сравнение этих двух подходов. Для достижения этой цели необходимо решить следующие **задачи**:

1. реализовать последовательный алгоритм построения минимальной выпуклой оболочки для компонент бинарного изображения на языке C++;
2. реализовать параллельный алгоритм построения минимальной выпуклой оболочки для компонент бинарного изображения на языке C++ с использованием технологии MPI;
3. реализовать ряд тестов с использованием фреймворка Google Test для проверки корректности работы программ;
4. провести вычислительные эксперименты для сравнения времени работы последовательного и параллельного подходов на разных входных данных и разном числе процессов;
5. сделать выводы об эффективности и качестве каждого подхода.

3 Описание алгоритма

Весь процесс построения выпуклой оболочки для компонент бинарного изображения можно разделить на **2 этапа**:

1. выделение компонент бинарного изображения;
2. построение выпуклой оболочки.

3.1 Выделение компонент бинарного изображения

На данном этапе каждому пикселю каждой компоненты присваивается номер. Тем самым происходит идентификация каждой компоненты, то есть отделение их друг от друга.

В данной лабораторной работе процесс выделения компонент реализован с помощью циклического алгоритма, использующего очередь задач.

3.2 Построение выпуклой оболочки

После завершения первого этапа образуется набор компонент. Однако изображение чаще всего имеет достаточно плотную структуру, поэтому в целях оптимизации компоненты дополнительно приводятся к более простому виду. Из рассмотрения удаляются точки, которые имеют более двух ненулевых соседей, а также те, которые имеют двух ненулевых соседей на одной линии (справа и слева или сверху и снизу от текущего пикселя). Данная процедура позволяет уменьшить количество рассматриваемых точек до минимума.

Для построения выпуклой оболочки для каждой компоненты бинарного изображения был выбран алгоритма Грэхема. На вход данному алгоритму подается массив точек, которые принадлежат конкретной компоненте.

Работу данного алгоритма можно разделить на несколько **этапов**:

1. выбираем точку из массива с минимальной x-координатой (если таких несколько, берем самую верхнюю из них), добавляем ее в ответ;
2. упорядочиваем оставшиеся точки по углу, который они образуют с выбранной точкой;
3. добавляем в ответ p_1 - первую точку из упорядоченного списка;
4. берем следующую точку k, и пока k и две предыдущие точки в текущей оболочке (p_i и p_{i-1}) образуют векторы (p_ik и p_ip_{i-1}), которые лежат не по часовой стрелке, исключаем из оболочки p_i;

5. добавляем в оболочку точку k;

6. повторяем пункты 4-5, пока есть точки.

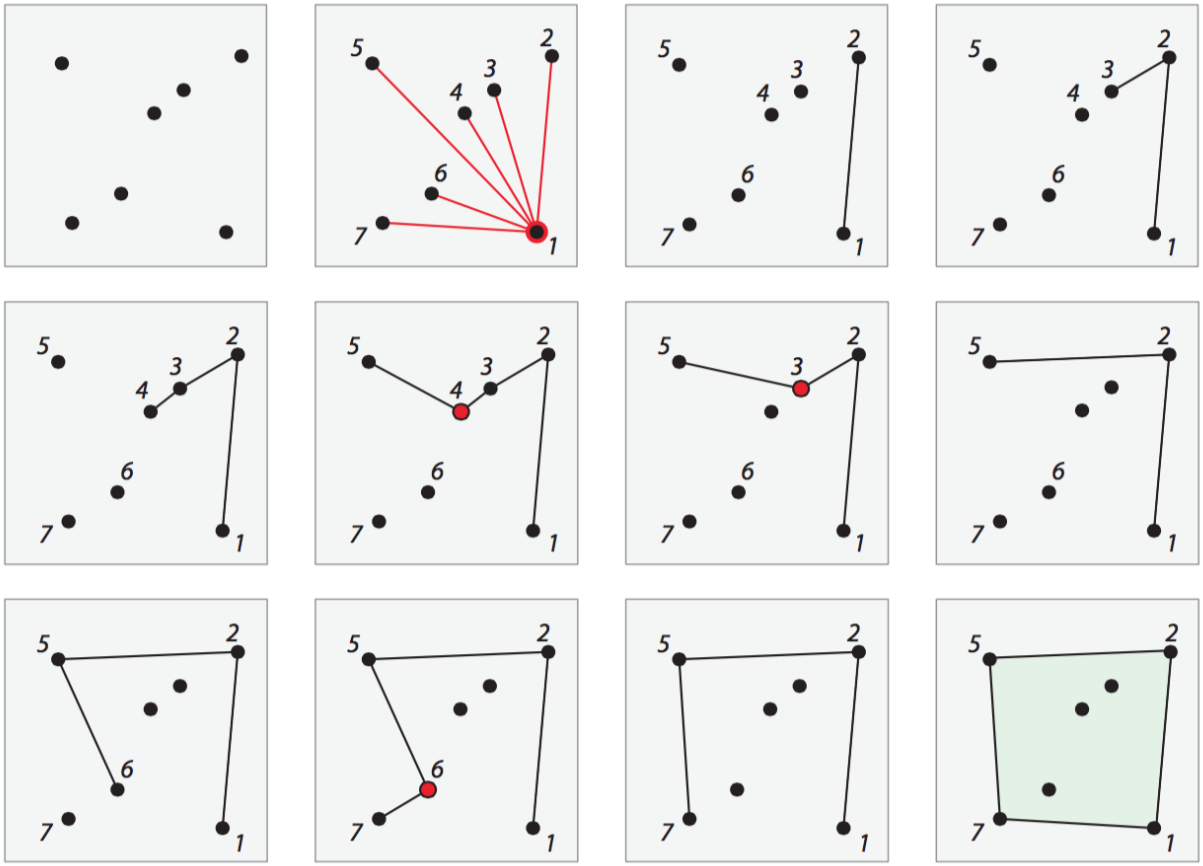

Рисунок 1. Алгоритм Грэхема в действии. Красные точки образуют левый поворот и удаляются из оболочки.

4 Описание схемы распараллеливания

В последовательной версии алгоритма второй этап происходит независимо для каждой компоненты. Это замечание можно использовать при реализации параллельной версии алгоритма.

Таким образом, каждый процесс получает свою часть компонент (равномерно распределенных), для которых он находит точки, составляющие выпуклые оболочки. Затем результаты, полученные в каждом процессе, отправляются в нулевой процесс, где они собираются в один массив, содержащий координаты точек выпуклых оболочек для каждой компоненты.

5 Программная реализация

Программа состоит из заголовочного файла $chbi.h$ и двух исходных файлов $chbi.cpp$ и $main.cpp$.

В заголовочном файле определены функции, реализующие последовательный и параллельный подходы для нахождения выпуклой оболочки произвольного множества точек, а также вспомогательные функции: для выделения компонент изображения, для подсчета количества компонент, для сокращения числа точек, генерации примеров и другие.

5.1 Функции для выделения компонент изображения

```
std::vector<int> findComponents(const std::vector<std::vector<int>>& image, int
↪  width, int height);
```

Функция $findComponents$ является не более чем общим интерфейсом. Она принимает на вход исходное изображение в виде двумерного вектора, его высоту и ширину. Весь процесс выполнения данной функции связан с вызовом для каждого пикселя процедуры $floodFill$. На выходе функция $findComponents$ возвращает промаркированное изображение в виде вектора.

```
void floodFill(std::vector<int>* image, int height, int width, int yStart, int
↪  xStart, int label);
```

Процедура $floodFill$ выполняет основную работу, а именно принимает на вход изображение в виде указателя на вектор, его высоту и ширину, точку старта, с которой необходимо начать маркировку и метку, которая присваивается данной компоненте. Данная функция ничего не возвращает, так как изменяет изображение, переданное ей в качестве аргумента.

5.2 Функции подсчета

```
int findCountComponents(const std::vector<int> &image);
```

Функция $findCountComponents$ используется для подсчета числа компонент в изображении. На вход принимается изображение в виде ссылки на вектор, а на выходе выдается число компонент в нем. Работает по принципу поиска компоненты с наибольшим номером метки.

```
int findCountPointsInComponent(const std::vector<int> &image);
```

Функция $findCountPointsInComponent$ используется для подсчета числа точек в компоненте. На вход принимается компонента (в виде части изображения), а на выходе выдается число точек в ней. Работает по принципу подсчета ненулевых пикселей.

5.3 Функция удаления лишний точек компоненты

```
std::vector<int> removeExtraPoints(const std::vector<int> &image, int width, int
↪  height, int component);
```

Функция $removeExtraPoints$ принимает на вход изображение в виде ссылки на вектор, его высоту и ширину, а также номер компоненты, которую необходимо оставить. На выходе получаем изображение в виде вектора, в котором число компонент уменьшено до минимума, что позволяет быстрее строить выпуклую оболочку.

5.4 Функция алгоритма Грэхема

```
void sort(std::vector<int>* points, int xMin, int yMin);
```

Функция $sort$ является вспомогательной. Она сортирует точки в процессе работы алгоритма Грэхема. На вход функции подается вектор точек, а также координаты точки, относительно которой нужно выполнить сортировку. На выходе получаем отсортированный по нужному правилу вектор.

```
int cross(int x1, int y1, int x2, int y2, int x3, int y3);
```

Функция $cross$ также является вспомогательной. С помощью нее определяется векторное произведение. На вход подаются координаты точек, а на выходе получаем их векторное произведение.

```
std::vector<int> graham(std::vector<int> points);
```

Функция $graham$ является основной, поскольку реализует алгоритм Грэхема. Она принимает на вход вектор точек. На выходе данная функция возвращает вектор точек, входящих в выпуклую оболочку.

5.5 Функция, реализующая последовательный подход

```
std::vector<int> getConvexHullSeq(const std::vector<std::vector<int>> &image, int
↪  width, int height);
```

Данная функция принимает на вход изображение в виде ссылки на вектор, его высоту и ширину. В процессе выполнения функции находятся его компоненты, и для каждой из них строится выпуклая оболочка. На выходе получаем вектор точек, которые образуют выпуклые оболочки для каждой компоненты.

5.6 Функция, реализующая параллельную версию

```
std::vector<int> getConvexHullPar(const std::vector<std::vector<int>> &image, int
↪  width, int height);
```

Данная функция принимает на вход изображение в виде ссылки на вектор, его высоту и ширину. В процессе выполнения функции находятся его компоненты, и для каждой из них строится выпуклая оболочка. На выходе получаем вектор точек, которые образуют выпуклые оболочки для каждой компоненты.

5.7 Функция заполнения изображения

```
void fillImageRandom(std::vector<std::vector<int>>* image, int width, int height);
```

Функция $fillImageRandom$ заполняет изображение единицами и нулями. На вход она принимает пустое изображение в виде ссылки на двумерный вектор, его высоту и ширину. Эта функция ничего не возвращает, поскольку изменяет непосредственно принятое изображение.

6 Результаты экспериментов

Прежде чем приступить к проведению экспериментов я проверил корректность работы последовательного и параллельного алгоритмов. Для этого с помощью фреймворка Google Test мной было разработано 5 тестов, каждый из которых, так или иначе, проверяет работоспособность программы при определенных, изначально заданных условиях. Успешное прохождение всех тестов доказывает корректность работы программы.

Для сравнения производительности последовательной и параллельной версий алгоритма мной были проведены эксперименты на разных размерах входных данных и на разном количестве процессов. При проведении экспериментов использовалась система со следующими **характеристиками**:

- Процессор: AMD Ryzen 5 4600H, 3.00 ГГц.
- Оперативная память: 16 ГБ, DDR4
- Операционная система: Windows 10

По результатам экспериментов был построен график (рис. 2) зависимости числа точек в изображении от времени, которое тратит та или иная версия алгоритма на его обработку.

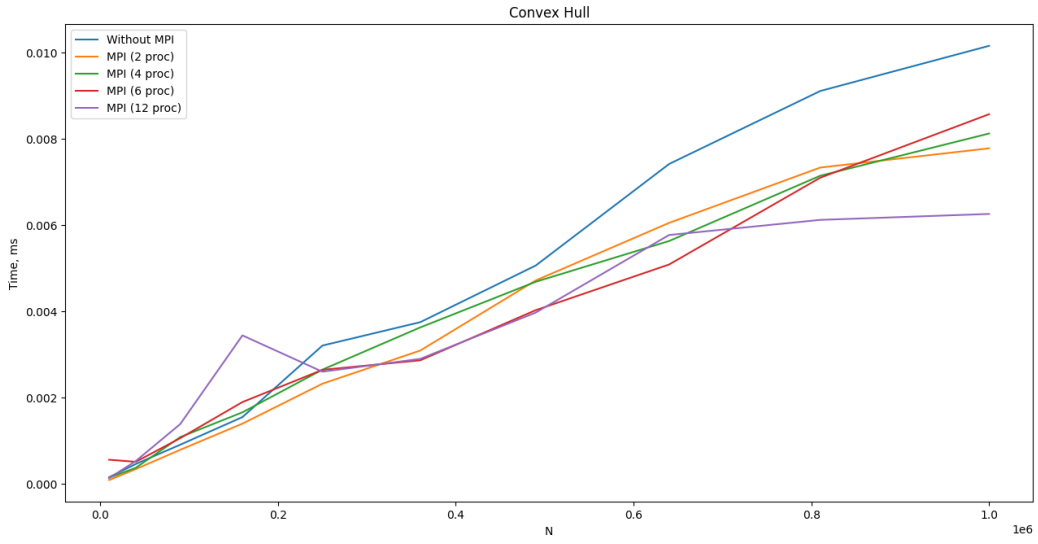

Рисунок 2. Результаты экспериментов

Также, в ходе эксперимента была сделана выборка нескольких контрольных замеров, результаты которых были сведены в таблицу (табл. 1).

Таблица 1. Контрольные значения экспериментов

Количество точек, N	**Время, мс**				
	Без MPI	**С использованием MPI**			
		2 процесса	**4 процесса**	**6 процессов**	**12 процессов**
100	0.003168	0.119753	0.157473	0.18108	0.0737149
10000	0.16032	0.09476	0.147707	0.565167	0.140087
90000	0.9138324	0.79558	1.08949	1.06523	1.39128
250000	3.210612	2.32737	2.65317	2.64831	2.60441
490000	5.06592	4.71992	4.68895	4.0314	3.97537
810000	9.108408	7.33513	7.14547	7.09905	6.12253
1000000	10.155588	7.78081	8.12407	8.57117	6.26017

7 Вывод

По графикам, полученным в ходе проведения экспериментов, можно сделать вывод о том, что параллельный алгоритм работает быстрее, чем последовательный.

Да, можно заметить, что при небольшом количестве точек в изображении параллельный алгоритм, особенно если число процессов велико, проигрывает в производительности последовательному, но это происходит потому что большая часть времени уходит на накладные расходы (создание процессов, пересылка данных между ними).

Однако, при росте размера входного изображения и при увеличении числа процессов можно заметить ускорение параллельного алгоритма относительно последовательного. Это означает, что чем больше размер исходного набора точек и чем больше число процессов, тем более быстро будет работать параллельный алгоритм относительно последовательного.

К последнему предложению стоит сделать небольшое замечание, касающееся того, что при очень большом числе процессов параллельный алгоритм может потерять свое ускорение, поскольку, как было сказано ранее, увеличиваются накладные расходы. Поэтому к выбору числа процессов тоже нужно подходить осторожно.

8 Заключение

В данной лабораторной работе мы реализовали алгоритм построения минимальной выпуклой оболочки для компонент бинарного изображения с использованием последовательного и параллельного подходов.

Наша основная цель была создать эффективный параллельный алгоритм, который бы работал быстрее, чем последовательный, при использовании нескольких процессов.

Мы провели ряд тестов, чтобы проверить корректность наших алгоритмов и сравнить их производительность. Тесты показали, что наши алгоритмы правильно вычисляют оболочку для различных входных данных. Кроме того, мы получили положительные результаты вычислительных экспериментов, которые подтверждают, что наш параллельный алгоритм дает значительное ускорение по сравнению с последовательным.

Таким образом, мы успешно выполнили поставленные задачи и достигли желаемого результата.

Список литературы

[1] Страуструп Б. Язык программирования C++. 4-е издание. / Б. Страуструп. – М.: «Бином», 2023. – 1216 с. - ISBN 978-5-6045724-6-7.

[2] Кормен, Т.Х. Алгоритмы: построение и анализ, 3-е издание. / Т.Х. Кормен, Ч.И. Лейзерсон, К. Штайн, Р.Л. Ривест. – М.: «Вильямс», 2013. – 1328 с. - ISBN 978-5-8459-1794-2.

[3] Официальный сайт библиотеки интерфейса передачи сообщений OpenMPI. - Режим доступа: https://www.open-mpi.org

Приложение

chbi.h

```
// Copyright 2023 Volodin Evgeniy
#ifndef TASKS_TASK_3_VOLODIN_E_CONVEX_HULL_BINARY_IMAGE_CHBI_H_
#define TASKS_TASK_3_VOLODIN_E_CONVEX_HULL_BINARY_IMAGE_CHBI_H_
#include <mpi.h>
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <utility>
#include <cmath>
#include <string>
#include <map>
#include <functional>
#include <random>

struct point2d {
        int x, y;
        point2d(int x, int y) : x(x), y(y) {}
};

std::vector<int> findComponents(const std::vector<std::vector<int>>& image, int
↪  width, int height);
void floodFill(std::vector<int>* image, int height, int width, int yStart, int
↪  xStart, int label);
int findCountComponents(const std::vector<int> &image);
int findCountPointsInComponent(const std::vector<int> &image);
std::vector<int> removeExtraPoints(const std::vector<int> &image, int width, int
↪  height, int component);
void sort(std::vector<int>* points, int xMin, int yMin);
std::vector<int> graham(std::vector<int> points);
std::vector<int> getConvexHullSeq(const std::vector<std::vector<int>> &image, int
↪  width, int height);
std::vector<int> getConvexHullPar(const std::vector<std::vector<int>> &image, int
↪  width, int height);
void fillImageRandom(std::vector<std::vector<int>>* image, int width, int height);

#endif  // TASKS_TASK_3_VOLODIN_E_CONVEX_HULL_BINARY_IMAGE_CHBI_H_
```

chbi.cpp

```cpp
// Copyright 2023 Volodin Evgeniy
#include "task_3/volodin_e_convex_hull_binary_image/chbi.h"
#include <boost/mpi/communicator.hpp>
#include <boost/mpi/collectives.hpp>

void floodFill(std::vector<int>* image, int height, int width, int yStart, int
↪  xStart, int label) {
        std::queue<point2d> tasks;
        tasks.push(point2d(xStart, yStart));
        while (!tasks.empty()) {
                int x = tasks.front().x;
                int y = tasks.front().y;
                tasks.pop();
                if (x >= 0 && y >= 0 && y < height && x < width && image->at(y *
                ↪  width + x) == 1) {
                        (*image)[y * width + x] = label;
                        tasks.push(point2d(x - 1, y - 1));
                        tasks.push(point2d(x - 1, y));
                        tasks.push(point2d(x - 1, y + 1));
                        tasks.push(point2d(x, y + 1));
                        tasks.push(point2d(x + 1, y + 1));
                        tasks.push(point2d(x + 1, y));
                        tasks.push(point2d(x, y - 1));
                        tasks.push(point2d(x + 1, y - 1));
                }
        }
}

std::vector<int> findComponents(const std::vector<std::vector<int>>& image, int
↪  width, int height) {
        std::vector<int> image_with_components(width * height);
        for (int i = 0; i < height; ++i) {
                for (int j = 0; j < width; ++j) {
                        image_with_components[i * width + j] = image[i][j];
                }
        }
        int label = 2;
        for (int i = 0; i < height; ++i) {
                for (int j = 0; j < width; ++j) {
```

```
                        if (image_with_components[i * width + j] == 1) {
                                floodFill(&image_with_components, height, width, i,
                                ↪  j, label);
                                ++label;
                        }
                }
        }
        for (int i = 0; i < height; ++i) {
                for (int j = 0; j < width; ++j) {
                        if (image_with_components[i * width + j] != 0) {
                                image_with_components[i * width + j]--;
                        }
                }
        }
        return image_with_components;
}

int findCountComponents(const std::vector<int> &image) {
        int count_components = 0;
        for (int i = 0; i < image.size(); ++i) {
                if (image[i] > count_components) {
                        count_components = image[i];
                }
        }
        return count_components;
}

int findCountPointsInComponent(const std::vector<int> &image) {
        int count_points = 0;
        for (int i = 0; i < image.size(); ++i) {
                if (image[i] != 0) {
                        count_points++;
                }
        }
        return count_points;
}

std::vector<int> removeExtraPoints(const std::vector<int> &image, int width, int
↪  height, int label) {
        std::vector<int> local_image(image);
```

```
for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
                if (image[i * width + j] == label) {
                        if ((j > 0) && (j < width - 1)) {
                                if ((i == 0) || (i == height - 1)) {
                                        if ((image[i * width + j - 1] ==
                                        ↪  label) && (image[i * width + j +
                                        ↪  1] == label)) {
                                                local_image[i * width + j] =
                                                ↪  0;
                                        }
                                }
                                if ((i > 0) && (i < height - 1)) {
                                        if (((image[i * width + j - 1] ==
                                        ↪  label) && (image[i * width + j +
                                        ↪  1] == label)) ||
                                        ((image[(i + 1) * width + j] ==
                                        ↪  label) && (image[(i - 1) * width
                                        ↪  + j] == label))) {
                                                local_image[i * width + j] =
                                                ↪  0;
                                        }
                                }
                                continue;
                        }
                        if ((i > 0) && (i < height - 1)) {
                                if ((j == 0) || (j == width - 1)) {
                                        if ((image[(i - 1) * width + j] ==
                                        ↪  label) &&
                                        (image[(i + 1) * width + j] ==
                                        ↪  label)) {
                                                local_image[i * width + j] =
                                                ↪  0;
                                        }
                                }
                                if ((j > 0) && (j < width - 1)) {
                                        if (((image[i * width + j - 1] ==
                                        ↪  label) && (image[i * width + j +
                                        ↪  1] == label)) ||
```

```
                                        ((image[(i + 1) * width + j] ==
                                        ↪  label) && (image[(i - 1) * width
                                        ↪  + j] == label))) {
                                                local_image[i * width + j] =
                                                ↪  0;
                                        }
                                }
                        }
                } else {
                        local_image[i * width + j] = 0;
                }
        }
    }
    int size = findCountPointsInComponent(local_image);
    std::vector<int> points(size * 2);
    int k = 0;
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            if (local_image[i * width + j] != 0) {
                points[k] = j;
                k++;
                points[k] = i;
                k++;
            }
        }
    }
    return points;
}

int cross(int x1, int y1, int x2, int y2, int x3, int y3) {
    return ((x2 - x1) * (y3 - y2) - (x3 - x2) * (y2 - y1));
}

void sort(std::vector<int>* points, int xMin, int yMin) {
    int size = points->size() / 2;
    for (int i = 1; i < size; ++i) {
        int j = i;
        while ((j > 0) && (cross(xMin, yMin, (*points)[2 * j - 2],
        (*points)[2 * j - 1], (*points)[2 * j], (*points)[2 * j + 1]) < 0)) {
            int temp = (*points)[2 * j - 2];
```

```
                        (*points)[2 * j - 2] = (*points)[2 * j];
                        (*points)[2 * j] = temp;
                        temp = (*points)[2 * j - 1];
                        (*points)[2 * j - 1] = (*points)[2 * j + 1];
                        (*points)[2 * j + 1] = temp;
                        j--;
                }
        }
}

std::vector<int> graham(std::vector<int> points) {
        std::vector<int> result;
        int num_points = points.size() / 2;
        if (num_points > 1) {
                int x_min = points[0];
                int y_min = points[1];
                int min_index = 0;
                for (int i = 2; i < points.size(); i+=2) {
                        if (points[i] < x_min || (points[i] == x_min && points[i + 1]
                        ↪  < y_min)) {
                                x_min = points[i];
                                y_min = points[i + 1];
                                min_index = i;
                        }
                }
                int temp = points[min_index];
                points[min_index] = points[num_points * 2 - 2];
                points[num_points * 2 - 2] = temp;
                temp = points[min_index + 1];
                points[min_index + 1] = points[num_points * 2 - 1];
                points[num_points * 2 - 1] = temp;
                points.pop_back();
                points.pop_back();
                sort(&points, x_min, y_min);
                result.push_back(x_min);
                result.push_back(y_min);
                result.push_back(points[0]);
                result.push_back(points[1]);
                for (int i = 2; i < points.size(); i += 2) {
                        int result_size = result.size();
```

```
                        int x1 = result[result_size - 4];
                        int y1 = result[result_size - 3];
                        int x2 = result[result_size - 2];
                        int y2 = result[result_size - 1];
                        int x3 = points[i];
                        int y3 = points[i + 1];

                        int rot = cross(x1, y1, x2, y2, x3, y3);
                        if (rot == 0) {
                                result[result_size - 2] = x3;
                                result[result_size - 1] = y3;
                        } else if (rot < 0) {
                                while (cross(result[(result.size()) - 4],
                                ↪  result[(result.size()) - 3],
                                result[(result.size()) - 2], result[(result.size()) -
                                ↪  1], x3, y3) < 0)
                                result.pop_back(), result.pop_back();
                                result.push_back(x3);
                                result.push_back(y3);
                        } else {
                                result.push_back(x3);
                                result.push_back(y3);
                        }
                }
        } else {
                result.resize(2);
                result[0] = points[0];
                result[1] = points[1];
        }
        return result;
}

std::vector<int> getConvexHullSeq(const std::vector<std::vector<int>> &image, int
↪  width, int height) {
        std::vector<int> convex_hull;
        std::vector<int> local_image = findComponents(image, width, height);
        int count_components = findCountComponents(local_image);
        for (int i = 1; i <= count_components; ++i) {
                std::vector<int> points = removeExtraPoints(local_image, width,
                ↪  height, i);
```

```
                std::vector<int> ch = graham(points);
                for (int j = 0; j < ch.size(); ++j) {
                        convex_hull.push_back(ch[j]);
                }
                convex_hull.push_back(-1);
        }
        return convex_hull;
}

std::vector<int> getConvexHullPar(const std::vector<std::vector<int>> &image, int
↪  width, int height) {
        std::vector<int> convex_hull;

        boost::mpi::communicator world;
        std::vector<int> local_image(width * height);
        int count_components;

        if (world.rank() == 0) {
                local_image = findComponents(image, width, height);
                count_components = findCountComponents(local_image);
                for (int i = 1; i < world.size(); ++i) {
                        world.send(i, 0, &count_components, 1);
                }
        } else {
                world.recv(0, 0, &count_components, 1);
        }

        const int proc_num = std::min(world.size(), count_components);
        if (world.rank() >= proc_num) {
                return std::vector<int>(0);
        }

        std::vector<int> components(count_components);
        for (int i = 0; i < count_components; ++i) {
                components[i] = i + 1;
        }

        int chunk = count_components / proc_num;
        int rem = count_components % proc_num;
        if (world.rank() < rem) {
```

```
        chunk++;
}

std::vector<int> local_components(chunk);

if (world.rank() == 0) {
        for (int i = 1; i < proc_num; ++i) {
                world.send(i, 0, local_image.data(), width * height);
        }
} else {
        world.recv(0, 0, local_image.data(), width * height);
}
if (world.rank() != 0) {
        world.send(0, 0, &chunk, 1);
}
if (world.rank() == 0) {
        int index = 0;
        for (int i = 1; i < proc_num; ++i) {
                int proc_chunk;
                world.recv(i, 0, &proc_chunk, 1);
                world.send(i, 0, components.data() + index, proc_chunk);
                index += proc_chunk;
        }
        for (int i = index, j = 0; i < count_components; ++i, ++j) {
                local_components[j] = components[i];
        }
} else {
        world.recv(0, 0, local_components.data(), chunk);
}
std::vector<int> local_convex_hull;
for (int i = 0; i < local_components.size(); ++i) {
        std::vector<int> points = removeExtraPoints(local_image, width,
        ↪  height, local_components[i]);
        std::vector<int> ch = graham(points);
        for (int j = 0; j < ch.size(); ++j) {
                local_convex_hull.push_back(ch[j]);
        }
        local_convex_hull.push_back(-1);
}
if (world.rank() == 0) {
```

```
                int size = 2 * findCountPointsInComponent(local_image);
                size += findCountComponents(local_image);
                for (int i = 1; i < proc_num; ++i) {
                        std::vector<int> buffer(size);
                        world.recv(i, 0, buffer.data(), size);
                        for (int i = 0; i < size; ++i) {
                                if (buffer[i] == -1 && buffer[i+1] == 0) {
                                        convex_hull.push_back(buffer[i]);
                                        break;
                                }
                                convex_hull.push_back(buffer[i]);
                        }
                }
                for (int i = 0; i < local_convex_hull.size(); ++i) {
                        convex_hull.push_back(local_convex_hull[i]);
                }
        } else {
                world.send(0, 0, local_convex_hull.data(), local_convex_hull.size());
        }
        return convex_hull;
}

void fillImageRandom(std::vector<std::vector<int>>* image, int width, int height) {
        std::default_random_engine rnd;
        std::uniform_int_distribution<> dist(0, 1);

        for (int i = 0; i < height; ++i) {
                for (int j = 0; j < width; ++j) {
                        int pixel = dist(rnd);
                }
        }
}
```

main.cpp

```
// Copyright 2023 Volodin Evgeniy
#include <gtest/gtest.h>
#include "task_3/volodin_e_convex_hull_binary_image/chbi.h"
#include <boost/mpi/environment.hpp>
#include <boost/mpi/communicator.hpp>
```

```
TEST(Parallel_Operations_MPI, Rectangular_Image) {
        boost::mpi::communicator world;
        const int width = 8, height = 5;
        std::vector<std::vector<int>> image({
                {0, 1, 0, 0, 0, 1, 1, 1},
                {1, 1, 0, 0, 1, 1, 1, 1},
                {1, 1, 0, 1, 1, 1, 1, 1},
                {1, 0, 0, 1, 0, 0, 0, 1},
                {1, 1, 0, 0, 0, 0, 1, 1}
        });

        std::vector<int> convex_hull_par = getConvexHullPar(image, width, height);

        if (world.rank() == 0) {
                std::vector<int> convex_hull_seq = getConvexHullSeq(image, width,
                ↪  height);

                for (int i = 0; i < convex_hull_par.size(); ++i) {
                        ASSERT_EQ(convex_hull_par[i], convex_hull_seq[i]);
                }
        }
}

TEST(Parallel_Operations_MPI, Square_Image) {
        boost::mpi::communicator world;
        const int width = 10, height = 10;
        std::vector<std::vector<int>> image = {
                {1, 1, 1, 0, 0, 0, 1, 1, 1, 1},
                {1, 1, 0, 0, 0, 0, 0, 1, 1, 1},
                {1, 0, 0, 0, 0, 0, 0, 0, 1, 1},
                {0, 0, 0, 0, 0, 0, 1, 0, 0, 1},
                {0, 0, 0, 1, 0, 1, 0, 0, 0, 0},
                {0, 0, 0, 0, 1, 1, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {1, 1, 1, 0, 0, 0, 0, 1, 1, 1},
                {1, 1, 1, 0, 0, 0, 1, 1, 1, 1},
                {1, 1, 1, 0, 0, 1, 1, 1, 1, 1},
        };

        std::vector<int> convex_hull_par = getConvexHullPar(image, width, height);
```

```
        if (world.rank() == 0) {
                std::vector<int> convex_hull_seq = getConvexHullSeq(image, width,
                ↪  height);

                for (int i = 0; i < convex_hull_par.size(); ++i) {
                        ASSERT_EQ(convex_hull_par[i], convex_hull_seq[i]);
                }
        }
}

TEST(Parallel_Operations_MPI, Random_Square_Image) {
        boost::mpi::communicator world;
        const int width = 12, height = 12;
        std::vector<std::vector<int>> image(height, std::vector<int>(width, 0));

        if (world.rank() == 0) {
                fillImageRandom(&image, width, height);
        }

        std::vector<int> convex_hull_par = getConvexHullPar(image, width, height);

        if (world.rank() == 0) {
                std::vector<int> convex_hull_seq = getConvexHullSeq(image, width,
                ↪  height);

                for (int i = 0; i < convex_hull_par.size(); ++i) {
                        ASSERT_EQ(convex_hull_par[i], convex_hull_seq[i]);
                }
        }
}

TEST(Parallel_Operations_MPI, Random_Rectangular_Image_Small) {
        boost::mpi::communicator world;
        const int width = 7, height = 8;
        std::vector<std::vector<int>> image(height, std::vector<int>(width, 0));

        if (world.rank() == 0) {
                fillImageRandom(&image, width, height);
        }
```

```
        std::vector<int> convex_hull_par = getConvexHullPar(image, width, height);

        if (world.rank() == 0) {
                std::vector<int> convex_hull_seq = getConvexHullSeq(image, width,
                ↪  height);

                for (int i = 0; i < convex_hull_par.size(); ++i) {
                        ASSERT_EQ(convex_hull_par[i], convex_hull_seq[i]);
                }
        }
}

TEST(Parallel_Operations_MPI, Random_Rectangular_Image_Big) {
        boost::mpi::communicator world;
        const int width = 35, height = 20;
        std::vector<std::vector<int>> image(height, std::vector<int>(width, 0));

        if (world.rank() == 0) {
                fillImageRandom(&image, width, height);
        }

        std::vector<int> convex_hull_par = getConvexHullPar(image, width, height);

        if (world.rank() == 0) {
                std::vector<int> convex_hull_seq = getConvexHullSeq(image, width,
                ↪  height);

                for (int i = 0; i < convex_hull_par.size(); ++i) {
                        ASSERT_EQ(convex_hull_par[i], convex_hull_seq[i]);
                }
        }
}

int main(int argc, char* argv[]) {
        boost::mpi::environment env(argc, argv);
        boost::mpi::communicator world;
        ::testing::InitGoogleTest(&argc, argv);
        ::testing::TestEventListeners& listeners =
        ↪  ::testing::UnitTest::GetInstance()->listeners();
```

```
        if (world.rank() != 0) {
                delete listeners.Release(listeners.default_result_printer());
        }
        RUN_ALL_TESTS();
        return 0;
}
```