ФГБОУ ВО «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»
Институт Информационных технологий, математики и механики
Фундаментальная информатика и информационные технологии

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«Построение выпуклой оболочки – проход Джарвиса.»

Выполнил:
Студент 3 курса, группы 3821Б1ФИ3:
Иванов Никита Антонович

Проверил:
Нестеров Александр Юрьевич

Нижний Новгород
2023

Содержание

1 Введение

1.1 Выпуклая оболочка

Выпуклой оболочкой множества X называется наименьшее выпуклое множество, содержащее X. «Наименьшее множество» здесь означает наименьший элемент по отношению к вложению множеств, то есть такое выпуклое множество, содержащее данную фигуру, что оно содержится в любом другом выпуклом множестве, содержащем данную фигуру.

Обычно выпуклая оболочка определяется для подмножеств векторного пространства над вещественными числами (в частности в евклидовом пространстве) и на соответствующих аффинных пространствах.

Выпуклая оболочка множества X обычно обозначается $\operatorname{Conv} X$.

Поиск выпуклой оболочки используется в обработке изображений, распознавании образов и разработке игр часто возникает задача поиска выпуклой оболочки на плоскости. Для этого существуют следующие алгоритмы:

1. Алгоритм обхода Джарвиса
2. Алгоритм обхода Грэхема
3. Алгоритм монотонных цепочек Эндрю
4. Алгоритм типа «Разделяй и властвуй»
5. Алгоритм «быстрого построения»
6. Алгоритм Чана

В данной лабораторной работе рассматривается алгоритм Джарвиса.

2 Постановка задачи

Цель работы: Написать паралельную версию алгоритма Джарвиса для поиска выпуклой оболочки изображения. Считается, что изображение задано в оттенках, то есть входные данные - двумерный массив байт, каждый байт соответствует пикселю изображения.

Описываемая работа содержит следующие задачи:

1. Написать алгоритм Джарвиса для поиска выпуклой оболочки и распареллить его.
2. Написать алгоритм биноризации изображения.
3. Написать тесты для проверки работоспособности.
4. Сравнить сихронный и паралельный алгоритм Джарвиса на производительность.
5. Сформулировать и обосновать вывод о том, в каких случаях целесообразно применять синхронный алгоритм, а в каких - параллельный.

Оборудование и программное обеспечение: Компьютер, поддерживающий работу Clion или Visual Studio Code.

3 Описание алгоритма

В этой лабораторной работе использовалось 3 алгоритма:

1. Алгоритм бинаризации изображения
2. Алгоритм Джарвиса для поиска выпуклой оболочки
3. Алгоритм проверки правильности решения

3.1 Алгоритм бинаризации изображения

Из условий поставленной задачи, изображение подается в виде двумерного массива. Каждый элемент из себя представляет целое число, принимающее значение от 0 до 255. Алгоритм бинаризации заключается в том, что если значение больше 178, то считаем это за точку. На выходе получаем список из точек, которые хранятся в структуре `std::pair<int, int>`, представляющую из себя пару координат точки.

3.2 Алгоритм Джарвиса для поиска выпуклой оболочки

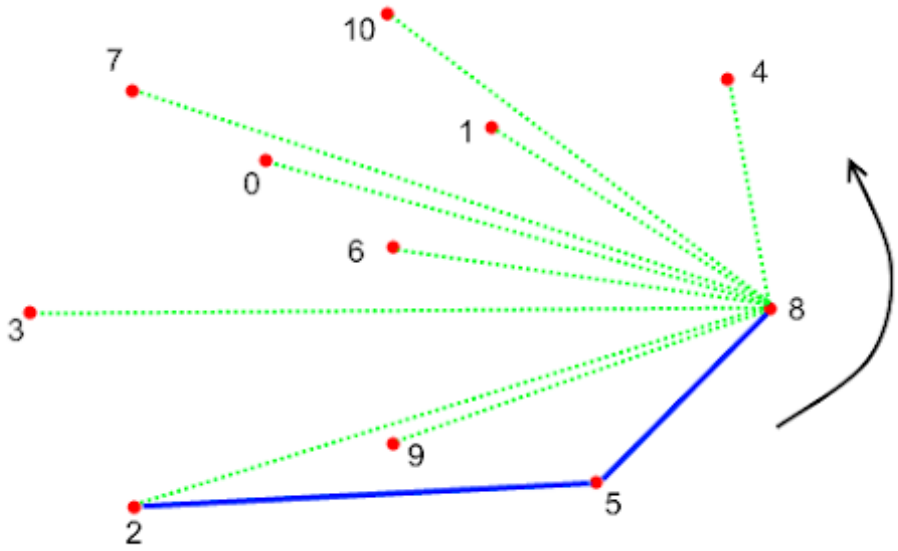

Рис. 1: Алгоритм Джарвиса

Пусть дан массив точек P и массив точек H. H - это массив, куда мы будем записывать точки, состовляющую выпуклую оболочку.

1. Необходимо найти точку, которая гарантировано будет входить в выпуклую оболочку; очевидно, что самая левая нижняя точка подходит под это условие. Находим точку с минимальной координатой x и y, и делаем ее текущей H[0].
2. Перебираем все оставшиеся точки двойным циклом по i и j. В массив H записываем самую правую точку относительно вектора H[i]P[j]. Для этого находим векторное

произведение, и если оно меньше 0, следовательно точка P[j] находится правее точки P[i] относительно H[i].

3. Алгоритм завершается тогда, когда в H будет записана стартовая вершина и следовательно линия замкнется.

3.3 Алгоритм проверки правильности решения

Алгоритм проверки правильности решения определяет пхождение каждой точки в выпуклую оболочку.

Пусть у нас есть множество P - множество всех точек, и множество H - множество точек, образующих выпуклую оболочку.

1. Поочереди перебираем каждую точку. Если выбранная точка является точкой, входящая в H, то пропускаем и идем дальше.

2. Если точка лежит на границе оболочки, то переходим на следующую итерацию.

2. Строим луч, который берет начало в выбранной точке, идет параллельно оси X и направлен от оси Y (то есть вправо).

3. Если луч пересекает границу оболочки нечетное количество раз, то точка внутри, иначе - вне оболочки.

4 Описание схемы распараллеливания

Ниже представлена схема распараллеливания.

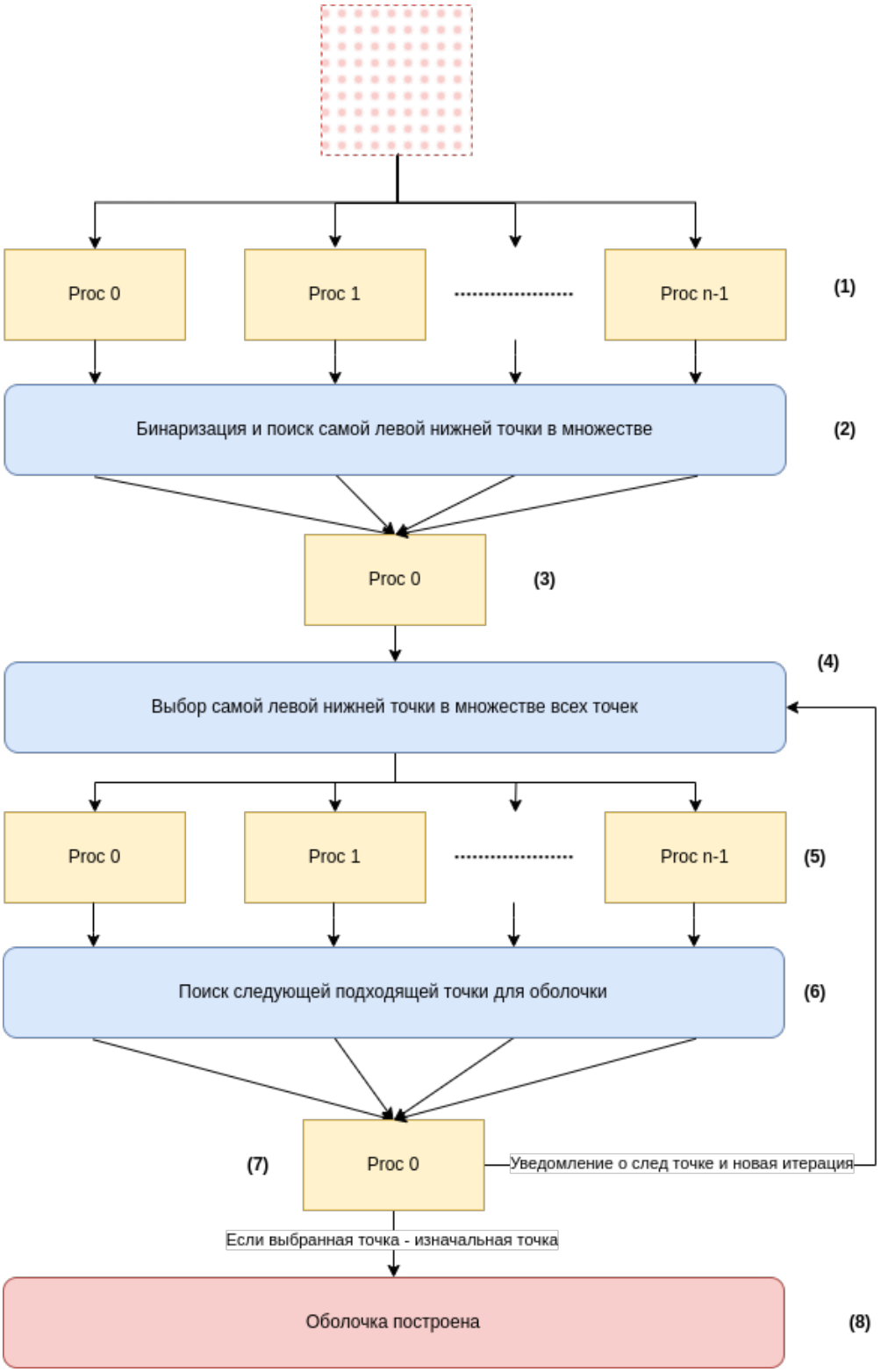

Рис. 2: Описание схемы распараллеливания

Описание каждого шага:

1. Исходное изображение делится между процессами при помощи `scatterv`. Остаток от деления достается нулевому процессу.

2. Каждый процесс получил свою часть изображения. В этой части каждыый процесс бинаризая изображение, тем самым получая множество P_i, где i - номер процесса. После каждый процесс в своем множестве находит самю нижнюю левую точку.

3. Все найденные точки собираются на нулевом процессе, используя `gather`. Нулевой процесс, назовем его **root**, собрал n точек, где n - кол-во процессов.

4. root процесс выбирает самую левую нижнюю точку, среди всего множества P. После рассылает остальным выбранную точку, при помощи `broadcast`.

5. Каждый процесс знает начальную точку. В пределах своего множества P_i ищет следующую подходящую точку ддля выпуклой оболочки, используя алгоритм Джарвиса.

6. Все выбранные точки собираются на root процессе через `gather`, и он выбирает среди них следующую в выпуклую оболочку.

7. Если новая точка это есть начальная точка, то заканчиваем поиск выпуклой оболочки. Иначе - рассылаем с помощью `broadcast` новую точку всем процессам и идем на новую итерацию алгоритма.

8. Алгоритм закончен, мы построили выпуклую оболочку. Все точки оболочки находятся в первом процессе.

5 Результаты экспериментов

5.1 Оборудование

Ноутбук

- Процессор AMD Ryzen 7 5700u with radeon graphics × 16
  - Количество ядер - 8
  - Количество потоков - 16
  - Базовая частота, в гигагерцах - 1.8
- RAM - 16ГБ
- OS Ubuntu 20.04.6 LTS

5.2 Описание эксперимента

В программе были поставлены счетчики времени, используя `boost::mpi::timer`. Запуск был осуществлен на 2, 3 и 4 процессах и 5 изображениях разных размеров. Для чистоты эксперимента собиралась статистика 20 запусков и выбиралось среднее значение. Результаты ниже в таблице.

Запуск алгоритма на 2 процессах		
Размер изображения ($n \times m$)	Параллельный алгоритм (сек)	Синхронный алгоритм (сек)
(9×11)	0.00014204	0.00001632
(100×100)	0.00025598	0.00047284
(200×300)	0.00123358	0.00290271
(400×500)	0.00462814	0.00976311
(500×600)	0.00618927	0.01366121

Запуск алгоритма на 3 процессах		
Размер изображения ($n \times m$)	Параллельный алгоритм (сек)	Синхронный алгоритм (сек)
(9×11)	0.00007188	0.00001284
(100×100)	0.00017434	0.00043404
(200×300)	0.00123358	0.00290271
(400×500)	0.00320425	0.00955258
(500×600)	0.00449706	0.01324535

Запуск алгоритма на 4 процессах		
Размер изображения ($n \times m$)	Параллельный алгоритм (сек)	Синхронный алгоритм (сек)
(9×11)	0.000192573	0.00001428
(100×100)	0.00015547	0.00047586
(200×300)	0.00068672	0.00273545
(400×500)	0.00223551	0.00896934
(500×600)	0.00295943	0.01321157

6 Анализ результатов

Посчитаем средние коэффициенты ускорения по формуле $p_i = \frac{\sum \frac{T_{si}}{T_{pi}}}{5}$, где T_{si} - время синхронного алгритма, а T_{pi} время параллельного алогритма для i задачи, где $i = 1, 2, ..., 5$.

Получаем следующие результаты:

Коэффициент ускорения	
Количество процессов	Ускорение
2	1.56
3	1.82
4	2.92

Вывод: ускорение находится в интервале от 1 до n.

Проверим ответ на логичность. Исходная сложность алгоритма - $O(N \cdot H)$, где N - общее количество точек, H - количество точек в выпуклой оболочке. Т.к. сложность больше, чем O(N), то полученный ответ вполне разумный.

7 Заключение

Параллельная версия реализации алгоритма бинаризации изображения и поиска выпуклой оболочки является более ыстрой, чем синхронная реализация. Но стоит иметь в виду, что при малых размерах изображения параллельная версия будет показывать более худшие результаты. Это связано из-за больших накладных расходов для взаимодействия между процессами.

8 Список Литературы

[1] Алгоритм Джарвиса (первый сайт). URL: https://ru.algorithmica.org/cs/convex-hulls/jarvis/

[2] Алгоритм Джарвиса (второй сайт). URL: https://habr.com/ru/articles/144921/

[3] Алгоритм Джарвиса (третий сайт). URL: https://cgraph.ru/node/327

[4] Выпуклая оболочка. URL: https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D0%BF%D1%83%D0%BA%D0%BB%D0%B0%D1%8F_%D0%BE%D0%B1%D0%BE%D0%BB%D0%BE%D1%87%D0%BA%D0%B0

[5] Boost MPI. URL: https://www.boost.org/doc/libs/1_61_0/doc/html/mpi/tutorial.html

[6] GigaChat: URL: https://developers.sber.ru/gigachat

[7] Сайт по созданию схем. URL: https://products.aspose.app/diagram/ru/flowchart

[8] GitHub. URL: https://github.com/Atikin-NT/ppc-2023-mpi

9 Приложение

```
std::vector<std::vector<int>> create_image(int n, int m) {
    std::random_device rd;
    std::uniform_int_distribution<int> unif(0, 255);
    std::vector<std::vector<int>> image(n, std::vector<int> (m));
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            image[i][j] = unif(rd);
    image[0][0] = 255;  // minimum one point

    return image;
}
```

Листинг 1: Создание изображения

```
std::vector<std::pair<int, int>> get_points_from_image(const std::vector<std::
    vector<int>> &image, int n) {
    boost::mpi::communicator world;

    int rank = world.rank();
    int delta = n / world.size();


    std::vector<std::pair<int, int>> points;
    for (int i = 0; i < image.size(); i++)
        for (int j = 0; j < image[0].size(); j++)
            if (image[i][j] > 178)
                points.emplace_back(j, rank * delta + i);
    return points;
}
```

Листинг 2: Бинаризация изображения

```
std::vector<P> JarvisParallel(const std::vector<std::vector<int>> &image, int n)
    {
    boost::mpi::communicator world;
    int rank = world.rank();
    int commsize = world.size();

    std::vector<P> result;
    std::vector<std::pair<int, int>> selected_points(commsize, std::make_pair(
   image[0].size(), image[0].size()));
    std::pair<int, int> start_point = std::make_pair(image[0].size(), image[0].
   size());

    std::vector<std::pair<int, int>> points = get_points_from_image(image, n);
    for (auto p : points)
        if (p.second < start_point.second || (p.second == start_point.second &&
   p.first < start_point.first))
            start_point = p;

    boost::mpi::gather(world, start_point, selected_points.data(), 0);
    if (rank == 0) {
        for (auto p : selected_points)
            if (p.second < start_point.second || (p.second == start_point.second
    && p.first < start_point.first))
                start_point = p;
    }
    boost::mpi::broadcast(world, start_point, 0);

    if (points.empty())
        points.emplace_back(start_point);

    std::pair<int, int> current = start_point;
    std::pair<int, int> next_point = points[0];

    while (true) {
        for (auto p : points) {
            if (p == current)
                continue;
            int val = crossProduct(current, next_point, p);
            if (val > 0) {
                next_point = p;
            } else if (val == 0) {
                if (distance(current, next_point, p) < 0)
                    next_point = p;
            }
        }

        boost::mpi::gather(world, next_point, selected_points.data(), 0);

        if (rank == 0) {
            for (auto p : selected_points) {
                if (p == current)
                    continue;
                int val = crossProduct(current, next_point, p);
                if (val > 0) {
                    next_point = p;
                } else if (val == 0) {
                    if (distance(current, next_point, p) < 0)
                        next_point = p;
                }
            }
```

```
            result.emplace_back(next_point);
        }
        boost::mpi::broadcast(world, next_point, 0);
        current = next_point;

        if (next_point == start_point)
            break;
    }
    return result;
}
```

Листинг 3: Параллельная версия Джарвиса

```
std::vector<P> Jarvis(std::vector<std::pair<int, int>> points) {
    std::pair<int, int> start_point = points[0];
    for (auto p : points)
        if (p.second < start_point.second || (p.second == start_point.second &&
   p.first < start_point.first))
            start_point = p;

    std::vector<P> result;
    std::pair<int, int> current = start_point;
    std::pair<int, int> next_point = points[0];
    while (true) {
        for (auto p : points) {
            if (p == current)
                continue;
            int val = crossProduct(current, next_point, p);
            if (val > 0) {
                next_point = p;
            } else if (val == 0) {
                if (distance(current, next_point, p) < 0)
                    next_point = p;
            }
        }

        result.emplace_back(next_point);
        current = next_point;

        if (next_point == start_point)
            break;
    }
    return result;
}
```

Листинг 4: Синхронная версия Джарвиса

```
bool inside_conv(const std::vector<P> &pol, std::vector<std::pair<int, int>>
    points) {
    int pol_size = pol.size();
    for (auto point_pair : points) {
        P point(point_pair);
        int j = pol_size - 1;
        bool res = false;
        for (int i = 0; i < pol_size; i++) {
            if (pol[i] == point || pol[j] == point || (pol[i].y == pol[j].y &&
    pol[i].y == point.y) ||
                (pol[i].x == pol[j].x && pol[i].x == point.x)) {
                res = true;
                break;
            }
            if ((pol[i].y < point.y && pol[j].y >= point.y || pol[j].y < point.y
    && pol[i].y >= point.y) &&
                (pol[i].x + (point.y - pol[i].y) / (pol[j].y - pol[i].y) * (pol[
    j].x - pol[i].x) == point.x)) {
                res = true;
                break;
            }
            if ((pol[i].y < point.y && pol[j].y >= point.y || pol[j].y < point.y
    && pol[i].y >= point.y) &&
                (pol[i].x + (point.y - pol[i].y) * (pol[j].x - pol[i].x) / (pol[
    j].y - pol[i].y) > point.x))
                res = !res;
            j = i;
        }
        if (!res)
            return false;
    }
    return true;
}
```

Листинг 5: Проверка вхождения точек в выпуклую оболочку