Лабораторная работа №4	M3137	2023
OpenMP	Кузнецов Сергей Павлович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: C++ компилятор (GCC) 12.2.0

Вариант Выбрана hard версия

Описание конструкций OpenMP для распараллеливания команд

Для распараллеливания команд использовались следующие конструкции:

#pragma omp parallel {} отвечает за объявление блока с параллельностью

#pragma omp for определяет итеративную конструкцию совместного использования работы, которая указывает, что итерации связанного цикла будут выполняться параллельно. Итерации цикла распределяются по потокам, которые уже существуют в команде, выполняющей параллельную конструкцию, с которой она связывается. Может работать с аргументами schedule(вид, [chunk_size]), nowait. Schedule определяет, как итерации цикла распределяются между потоками. Когда задан параметр schedule(static, chunk_size) итерации делятся на фрагменты размера, указанного параметром chunk_size. Блоки статически присваиваются потокам в команде циклическим образом в порядке номера потока. Если не указан размер фрагмента, пространство итераций делится на фрагменты, приблизительно равные по размеру, при этом каждому потоку назначается один фрагмент. Когда указан schedule(dynamic, chunk_size), итерации делятся на серию блоков, каждый из которых содержит итерации chunk_size. Каждый фрагмент присваивается потоку, который ожидает назначения. Поток выполняет фрагмент итераций, а затем ожидает своего следующего назначения, пока не останется фрагментов, которые нужно назначить. Последний назначаемый фрагмент может иметь меньшее количество итераций. Если не указан размер chunk_size, по умолчанию он равен 1. Nowait – после выполнения выделенного участка

происходит неявная барьерная синхронизация
параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки, если в подобной задержке нет необходимости, опция nowait позволяет потокам, уже дошедшим до конца участка, продолжить выполнение без синхронизации с остальными.

#pragma omp critical {} в этой секции одновременно может быть не больше одного потока.

omp_set_num_threads(count_thread) – определяет сколько тредов будет использоваться

Описание работы написанного кода

На вход программе подается количество тредов, которое будет использовано во время выполнения распараллеленного кода, имя входного файла, имя выходного файла. Из файла считываются значения в буфер, после файл закрывается. Из буфера считываются ширина и высота, и проверяется, что файл соответствуют нужному типу. Далее определяется индекс, с которого начинаются данные в загруженном буфере. За все это отвечает метод read(). Далее запускается метод пороговой фильтрации оцу для трех порогов, если количество тредов не отрицательно, запускается метод с распараллеленным кодом, иначе запускается метод без распараллеливания. В методах вычисляется гистограмма (количество пикселей каждого оттенка), рассмотрим распараллеленный вариант

```
    size_t shades[count_shades];
    for (int i = 0; i < 256; i++) {
        shades[i] = 0;
    }

#pragma omp parallel
    {
        size_t shades_local[256];
        for (int i = 0; i < 256; i++) {
            shades_local[i] = 0;
        }

#pragma omp for nowait
        for (size_t i = 0; i < count_char; i++) {
            shades_local[(unsigned char) int(buffer[i + start_index])]++;
        }

#pragma omp critical
        {
            for (int i = 0; i < 256; i++) {
                shades[i] += shades_local[i];
            }
```

```
        }
    }
```

Таблица 1 – распараллеливание подсчета гистограммы в методе otsu_with_parallel

Для начала задается массив с 0, далее начинается распараллеленный блок, в котором, создается локальный массив для каждого треда, так как если сразу добавлять значения в общий массив, то какое-то значение может потеряться, при одновременном использование несколькими потоками. Далее идет подсчет гистограммы из входных данных, записанных в буфере. После подсчета каждый тред добавляет свои значения в общий массив, для этого используется блок critical.

```
double max_sigma = 0;
int res[3];
size_t prefix_shades[256];
size_t prefix_mu[256];
prefix_shades[0] = shades[0];
prefix_mu[0] = 0;
for (int i = 1; i < 256; i++) {
    prefix_shades[i] = shades[i] + prefix_shades[i - 1];
    prefix_mu[i] = prefix_mu[i - 1] + i * shades[i];
}

#pragma omp parallel
{
    double max_sigma_local = max_sigma;
    int res_local[3];

    for (int f_1 = 0; f_1 < count_shades - 2; f_1++) {
        for (int f_2 = f_1 + 1; f_2 < count_shades - 1; f_2++) {
#pragma omp for nowait
            for (int f_3 = f_2 + 1; f_3 < count_shades; f_3++) {
                double q_double[4];
                double m_double[4];

                q_double[0] = (double) prefix_shades[f_1];
                q_double[1] = (double) (prefix_shades[f_2] - prefix_shades[f_1]);
                q_double[2] = (double) (prefix_shades[f_3] - prefix_shades[f_2]);
                q_double[3] = (double) (prefix_shades[255] - prefix_shades[f_3]);
                m_double[0] = (double) prefix_mu[f_1];
                m_double[1] = (double) (prefix_mu[f_2] - prefix_mu[f_1]);
                m_double[2] = (double) (prefix_mu[f_3] - prefix_mu[f_2]);
                m_double[3] = (double) (prefix_mu[255] - prefix_mu[f_3]);

                // m[i] = sum(i * P(i)) / count_char / q[i]
                // sigma = sum(m[i]^2) * q[i]
                // => sigma = sum(i * P(i) / count_char)^2 / q[i]
                // q[i] = sum(P(i)) / count_char
                // => sigma = sum(i * P(i))^2 / sum(P(i)) / count_char
                double sigma = 0;
                for (int i = 0; i < 4; i++) {
                    if (q_double[i] == 0) {
                        break;
```

```
                    }
                    sigma += m_double[i] * m_double[i] / q_double[i] / (double)
count_char;
                }
                if (sigma > max_sigma_local) {
                    max_sigma_local = sigma;
                    res_local[0] = f_1;
                    res_local[1] = f_2;
                    res_local[2] = f_3;
                }
            }
        }
    }
#pragma omp critical
        {
            if (max_sigma_local > max_sigma) {
                max_sigma = max_sigma_local;
                for (int i = 0; i < 3; i++) {
                    res[i] = res_local[i];
            }
        }
    }
}
```

Таблица 2 – метод оцу в методе otsu_with_parallel

Далее запускается метод оцу (полный перебор всех комбинаций порогов и выбор наилучшей из них). Предварительно подсчитаны префиксы, для оптимизации алгоритма. В блоке с параллельностью также созданы локальные массивы, для корректного исполнения. Параллельно выполняются итерации 3 вложенного цикла, в котором и происходит подсчёт значения межклассовой дисперсии и сравнения с локальным максимум. В комментариях кода кратко приведены математические формулы рассуждения, которые служили для оптимизации математических операций.

Далее происходит перезапись новых данных в буфер, также распараллеленная

```
#pragma omp parallel
    {
#pragma omp for
        for (size_t i = 0; i < count_char; i++) {
            int now = (unsigned char) buffer[i + start_index];
            if (now <= result[0]) {
                buffer[i + start_index] = (char) 0;
            } else if (now <= result[1]) {
                buffer[i + start_index] = (char) 84;
            } else if (now <= result[2]) {
                buffer[i + start_index]= (char) 170;
            } else {
                buffer[i + start_index] = (char) 255;
            }
```

```
            }
        }
```

Таблица 3 – перезапись буфера с новыми данными

После происходит запись выходной в файл.

Результат работы программы

```
Time (0 thread(s)): 9.00006 ms
77 130 187
```

Процессор Intel Core i9-129000H

Экспериментальная часть

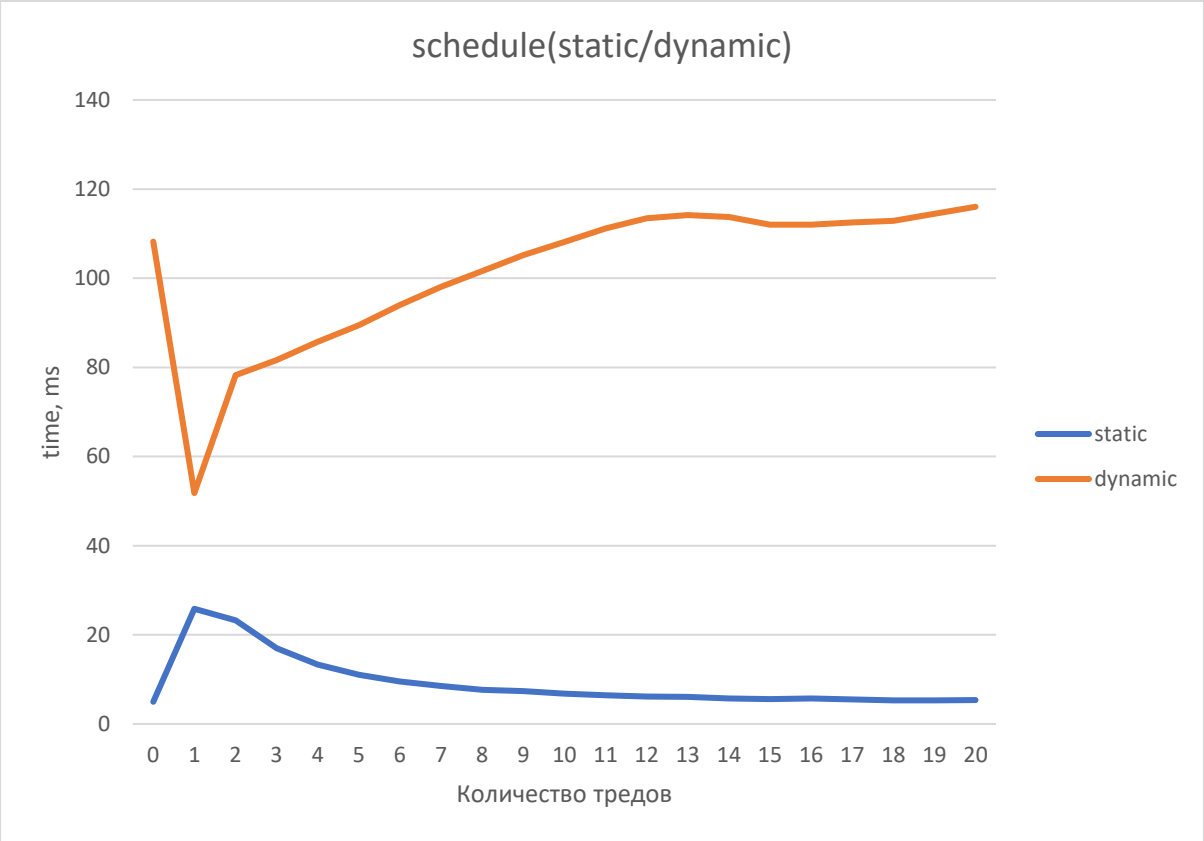

График 1 зависимость времени исполнения программы от количества тредов при использование schedule(static/dynamic) без аргументов

Из графика видно, что использование static дает больший выигрыш по времени, чем dynamic.

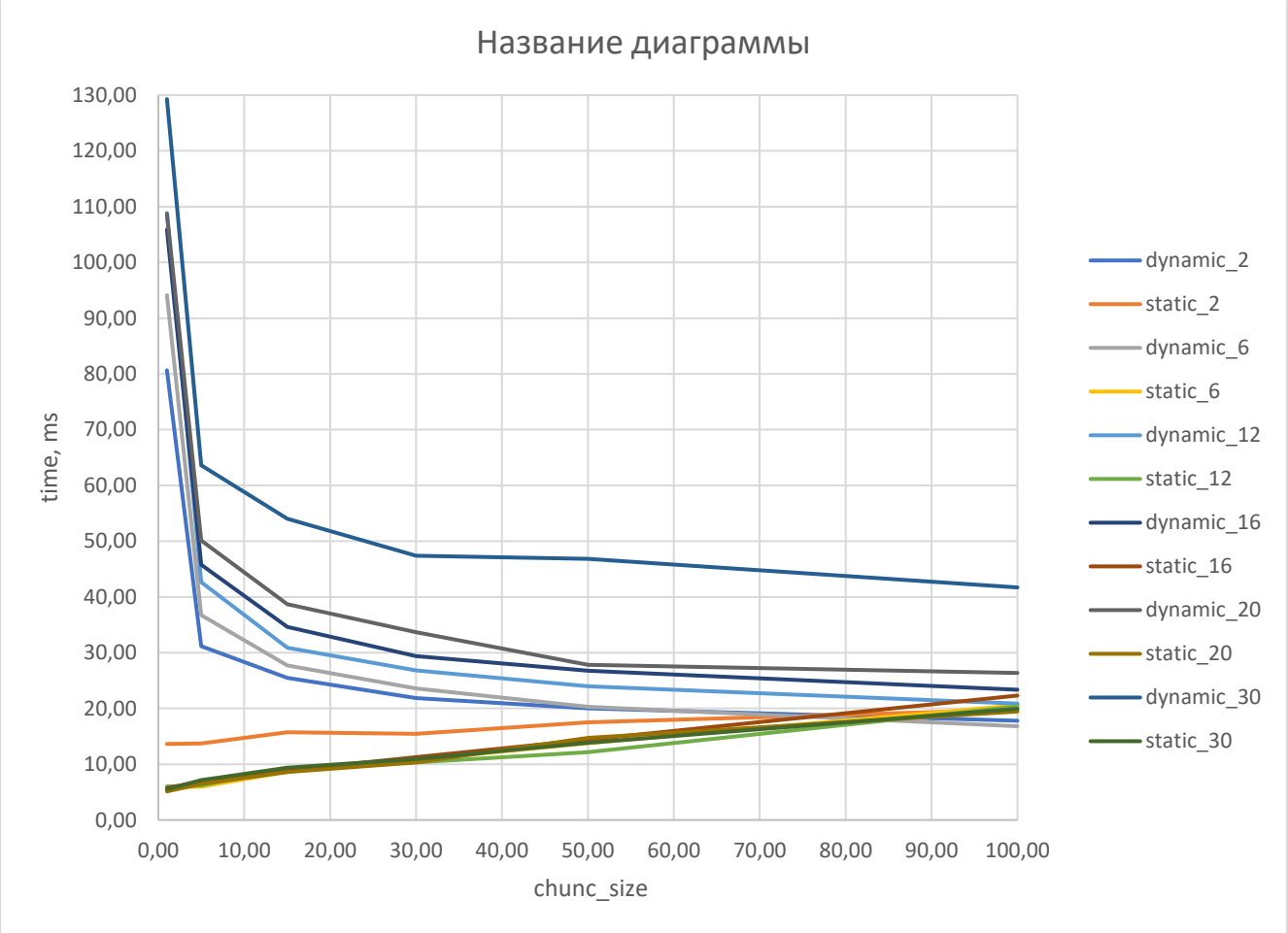

График 2 – зависимости от размера блока (chunk_size) в schedule при использовании различного вида и различном количестве тредов <вид_количество тредов>

На графике видно, что наилучший результат показывает static при количестве тредов > 2.

Количество тредов	Время исполнения, ms
-1 (без openmp)	18,845
1	24,44

Таблица 4 – время исполнения программы на 1 треде и без openmp

Из таблицы видно, что распараллеленный код работает с 1 тредов дольше, чем код без распараллеливания. Это происходит так, потому что создание треда тоже занимает время.

На данном коде static работает эффективнее чем dynamic, так как dynamic, особенно при малых значениях, тратит время на передачу новых команд, когда тред заканчивает обрабатывать уже выданные ему, и время сэкономленное на том, что треды не простаивают, если есть необработанные команды, незначительно, по сравнению с раздачей команд.

Список источников

https://learn.microsoft.com/en-us/cpp/parallel/openmp/2-directives?view=msvc-170#241-for-construct

https://ru.wikipedia.org/wiki/Метод_Оцу

Листинг кода

hard.cpp

```cpp
#include <iostream>
#include <omp.h>
#include <fstream>
#include <cmath>

using namespace std;

int result[3];
size_t width, height;
int count_shades = 255;
long long count_char = 0;
char *buffer;
int start_index;

void otsu_with_parallel(int number_thread) {
    if (number_thread > 0) {
        omp_set_num_threads(number_thread);
    }

    size_t shades[count_shades];
    for (int i = 0; i < 256; i++) {
        shades[i] = 0;
    }

#pragma omp parallel
    {
        size_t shades_local[256];
        for (int i = 0; i < 256; i++) {
            shades_local[i] = 0;
        }

#pragma omp for nowait
        for (size_t i = 0; i < count_char; i++) {
            shades_local[(unsigned char) int(buffer[i + start_index])]++;
        }

#pragma omp critical
        {
            for (int i = 0; i < 256; i++) {
                shades[i] += shades_local[i];
```

```
            }
        }
    }


    double max_sigma = 0;
    int res[3];
    size_t prefix_shades[256];
    size_t prefix_mu[256];
    prefix_shades[0] = shades[0];
    prefix_mu[0] = 0;
    for (int i = 1; i < 256; i++) {
        prefix_shades[i] = shades[i] + prefix_shades[i - 1];
        prefix_mu[i] = prefix_mu[i - 1] + i * shades[i];
    }

#pragma omp parallel
    {
        double max_sigma_local = max_sigma;
        int res_local[3];

        for (int f_1 = 0; f_1 < count_shades - 2; f_1++) {
            for (int f_2 = f_1 + 1; f_2 < count_shades - 1; f_2++) {
#pragma omp for nowait
                for (int f_3 = f_2 + 1; f_3 < count_shades; f_3++) {
                    double q_double[4];
                    double m_double[4];

                    q_double[0] = (double) prefix_shades[f_1];
                    q_double[1] = (double) (prefix_shades[f_2] -
prefix_shades[f_1]);
                    q_double[2] = (double) (prefix_shades[f_3] -
prefix_shades[f_2]);
                    q_double[3] = (double) (prefix_shades[255] -
prefix_shades[f_3]);

                    m_double[0] = (double) prefix_mu[f_1];
                    m_double[1] = (double) (prefix_mu[f_2] - prefix_mu[f_1]);
                    m_double[2] = (double) (prefix_mu[f_3] - prefix_mu[f_2]);
                    m_double[3] = (double) (prefix_mu[255] - prefix_mu[f_3]);

                    // m[i] = sum(i * P(i)) / count_char / q[i]
                    // sigma = sum(m[i]^2) * q[i]
                    // => sigma = sum(i * P(i) / count_char)^2 / q[i]
                    // q[i] = sum(P(i)) / count_char
                    // => sigma = sum(i * P(i))^2 / sum(P(i)) / count_char
                    double sigma = 0;
                    for (int i = 0; i < 4; i++) {
                        if (q_double[i] == 0) {
                            break;
                        }
                        sigma += m_double[i] * m_double[i] / q_double[i] / (double)
count_char;
                    }
                    if (sigma > max_sigma_local) {
                        max_sigma_local = sigma;
                        res_local[0] = f_1;
                        res_local[1] = f_2;
                        res_local[2] = f_3;
                    }
                }
```

```
            }
        }
#pragma omp critical
        {
            if (max_sigma_local > max_sigma) {
                max_sigma = max_sigma_local;
                for (int i = 0; i < 3; i++) {
                    res[i] = res_local[i];
                }
            }
        }
    }

    result[0] = res[0];
    result[1] = res[1];
    result[2] = res[2];


#pragma omp parallel
    {
#pragma omp for
        for (size_t i = 0; i < count_char; i++) {
            int now = (unsigned char) buffer[i + start_index];
            if (now <= result[0]) {
                buffer[i + start_index] = (char) 0;
            } else if (now <= result[1]) {
                buffer[i + start_index] = (char) 84;
            } else if (now <= result[2]) {
                buffer[i + start_index]= (char) 170;
            } else {
                buffer[i + start_index] = (char) 255;
            }
        }
    }
}

void otsu_without_parallel() {
    int shades[256];
    for (int i = 0; i < 256; i++) {
        shades[i] = 0;
    }
    for (int i = 0; i < count_char; i++) {
        shades[(unsigned char) int(buffer[i + start_index])]++;
    }

    double max_sigma = 0;
    int res[3];
    int prefix_shades[256];
    int prefix_mu[256];
    prefix_shades[0] = shades[0];
    prefix_mu[0] = 0;
    for (int i = 1; i < 256; i++) {
        prefix_shades[i] = shades[i] + prefix_shades[i - 1];
        prefix_mu[i] = prefix_mu[i - 1] + i * shades[i];
    }
    double max_sigma_local = max_sigma;
    int res_local[3];

    for (int f_1 = 0; f_1 < count_shades - 2; f_1++) {
        for (int f_2 = f_1 + 1; f_2 < count_shades - 1; f_2++) {
            for (int f_3 = f_2 + 1; f_3 < count_shades; f_3++) {
```

```
                double q_double[4];
                double m_double[4];

                q_double[0] = (double) prefix_shades[f_1];
                q_double[1] = (double) (prefix_shades[f_2] - prefix_shades[f_1]);
                q_double[2] = (double) (prefix_shades[f_3] - prefix_shades[f_2]);
                q_double[3] = (double) (prefix_shades[255] - prefix_shades[f_3]);

                m_double[0] = (double) prefix_mu[f_1];
                m_double[1] = (double) (prefix_mu[f_2] - prefix_mu[f_1]);
                m_double[2] = (double) (prefix_mu[f_3] - prefix_mu[f_2]);
                m_double[3] = (double) (prefix_mu[255] - prefix_mu[f_3]);

                // m[i] = sum(i * P(i)) / count_char / q[i]
                // sigma = sum(m[i]^2) * q[i]
                // => sigma = sum(i * P(i) / count_char)^2 / q[i]
                // q[i] = sum(P(i)) / count_char
                // => sigma = sum(i * P(i))^2 / sum(P(i)) / count_char
                double sigma = 0;
                for (int i = 0; i < 4; i++) {
                    if (q_double[i] == 0) {
                        break;
                    }
                    sigma += m_double[i] * m_double[i] / q_double[i] / (double)
count_char;
                }
                if (sigma > max_sigma_local) {
                    max_sigma_local = sigma;
                    res_local[0] = f_1;
                    res_local[1] = f_2;
                    res_local[2] = f_3;
                }
            }
        }
        if (max_sigma_local > max_sigma) {
            max_sigma = max_sigma_local;
            for (int i = 0; i < 3; i++) {
                res[i] = res_local[i];
            }
        }
    }

    result[0] = res[0];
    result[1] = res[1];
    result[2] = res[2];

    for (size_t i = 0; i < count_char; i++) {
        int now = (unsigned char) buffer[i + start_index];
        if (now <= result[0]) {
            buffer[i + start_index] = (char) 0;
        } else if (now <= result[1]) {
            buffer[i + start_index] = (char) 84;
        } else if (now <= result[2]) {
            buffer[i + start_index] = (char) 170;
        } else {
            buffer[i + start_index] = (char) 255;
        }
    }
}

void threshold_filtering_otsu_method(int count_thread) {
```

```
    if (count_thread >= 0) {
        otsu_with_parallel(count_thread);
    } else {
        otsu_without_parallel();
    }
}



int read(string name_in) {
    ifstream fin(name_in, ifstream::binary);

    if (!fin.is_open()) {
        cout << "Error, not open input file" << endl;
        return -1;
    } else {
        fin.seekg(0, fin.end);
        size_t length = fin.tellg();
        fin.seekg(0, fin.beg);

        buffer = new char[length];
        fin.read(buffer, length);

        string test = "";
        for (int i = 0; i < 3; i++) {
            test += buffer[i];
        }
        start_index = 3;
        if (test != "P5\n") {
            cout << "Error, file format is incorrect";
            return -1;
        }
        try {
            test = "";
            while (!isspace(buffer[start_index])) {
                test += buffer[start_index];
                start_index++;
            }
            start_index++; // ' '
            width = atoi(test.c_str());
            test = "";
            while (buffer[start_index] != '\n') {
                test += buffer[start_index];
                start_index++;
            }
            height = atoi(test.c_str());
            start_index++; // \n
            test = "";
            for (int i = 0; i < 4; i++) {
                test += buffer[start_index];
                start_index++;
            }

            if (test != "255\n") {
                cout << "Error, file format is incorrect";
                return -1;
            }

            if (length - start_index != width * height) {
                cout << "Error, file format is incorrect";
                return -1;
```

```
            }
        } catch (exception ex) {
            cout << "Error, file format is incorrect";
            return -1;
        }

        count_char = length - start_index;
        fin.close();

    }
    return 0;
}


void write(string name_out) {
    ofstream fout(name_out);
    fout << "P5\n" << width << " " << height << "\n255\n";
    for (size_t i = 0; i < count_char; i++) {
        fout << buffer[i + start_index];
    }
    fout.close();
}


int main(int argc, char *argv[]) {
    if (argc != 4) {
        cout << "Error, expect 4 arg, actual " << argc;
        return 0;
    }
    int count_threads = atoi(argv[1]);
    string name_in = argv[2];
    string name_out = argv[3];

    if (read(name_in) != 0) {
        return 0;
    }

    double time_start = omp_get_wtime();

    threshold_filtering_otsu_method(count_threads);

    double time_count = omp_get_wtime() - time_start;
    printf("Time (%i thread(s)): %g ms\n", count_threads, time_count * 1000);

    for (int i = 0; i < 3; i++) {
        cout << result[i] << " ";
    }
    cout << "\n";

    write(name_out);
//    cout << "Successful\n";

    return 0;
}
```