САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил: Трофимов Максим Владимирович

Номер ИСУ: 334948

студ. гр. M3135

Санкт-Петербург

2021

Цель работы: знакомство со стандартом OpenMP.

Инструментарий и требования к работе: C++, OpenMP.

Теоретическая часть

OpenMP

OpenMP — это библиотека для параллельного программирования вычислительных систем. Официально поддерживается C/C++ и Фортран. Библиотека OpenMP подходит только для программирования систем с общей памятью, при этом используется параллелизм потоков. Потоки создаются в рамках единственного процесса и имеют свою собственную память. Кроме того, все потоки имеют доступ к памяти процесса. Для того, чтобы создать зону, в которой будет происходить распараллеливание процесса, необходимо указать в коде “#pragma omp parallel {…your code…}”. При захождении в данную зону программа создает потоки (количество потоков можно указать в начале программы с помощью “omp_set_num_threads(n)” или в качестве аргумента “#pragma omp parallel num_threads(n)”) и в зависимости от задачи распределяет ее по потокам.

В качестве аргументов можно указать, какие переменные будут локальными (private) в данном блоке, а какие взяты извне (shared): “#pragma omp parallel shared(…) private(…)”.

Также существуют инструкции “#pragma omp critical” и “#pragma omp barrier”. Первая позволяет избавиться от конфликтов между потоками (если они, к примеру изменяют один и тот же участок памяти (например меняют значение в ячейке массива или увеличивают переменную (вообще для операций “+, *, -, /, &, ^, |, <<, >>” лучше использовать “#pragma omp atomic”, т.к он обычно быстрее))), приостанавливая действие других, пока в одном выполняется данный код. Второй же является так называемым «барьером», т.е. все потоки выполняются

параллельно и когда доходят до данной инструкции, то останавливаются и ждут остальные потоки.

Если существует несколько различных задач, не связанных между собой, то имеет смысл использовать инструкцию “#pragma omp parallel sections {…code…}”, где внутри фигурных скобок разделить код на несколько секций: “#pragma omp section {…code…}”. Таким образом все данные секции будут выполняться параллельно.

Наиболее часто встречающаяся возможность OpenMP – это распараллеливание цикла. Это можно сделать с помощью “#pragma omp for <args>” внутри зоны “#pragma omp parallel”. С помощью аргумента <schedule(type[, chunks])> можно указать как именно мы хотим параллелить. При <schedule(static)> итерации цикла будут поровну (приблизительно) поделены между потоками. Нулевой поток получит первые $\frac{n}{p}$ итераций, первый — вторые и т.д. При <schedule(static, k)> каждый поток получает заданное число итераций (k) в начале цикла, затем (если остались итерации) процедура распределения продолжается. Планирование выполняется один раз, при этом каждый поток «узнает» итерации которые должен выполнить. При <schedule(dynamic[, k])> каждый поток получает заданное число итераций (по умолчанию k=1), выполняет их и запрашивает новую порцию. В отличии от статического планирования выполняется многократно (во время выполнения программы).

Автоконтрастность

Описание алгоритма автоконтрастности:

Суть задачи была в том, чтобы растянуть значения каналов R, G, B (или одного канала для изображений .pgm) до отрезка [0; 255] при этом игнорируя некоторое количество самых темных/светлых значений. Алгоритм сам по себе совсем не сложный. Для начала необходимо найти минимум и максимум из всех каналов с учетом коэффициента шума. Т.е. находим то количество значений, которое

необходимо пропустить: $CountPropusk = h * w * coeff$. Затем находим k-ю порядковую статистику отдельно в каждом канале (тем самым получаем значения $minR, maxR, minG, maxG, minB, maxB$). Далее находим минимум из минимумов и максимум из максимумов и таким образом получаем значения $\min\_br, \max\_br$. Далее проходимся по всем пикселям и каждый канал меняем по несложной математической формуле из школы: $\frac{x - \min\_br}{\max\_br - \min\_br} * 255$.

Описание работы кода

Программа написана на C++17. Для запуска программы через консоль необходимо ввести: “filename.exe <кол-во потоков> <имя входного файла> <имя выходного файла> <коэффициент>”.

1. Открываем входной файл для записи с помощью std::ifstream. Читаем побайтово, определяем тип файла (P6/P5). Записываем все байты в массив типа pix (pix – структура, хранящая 3 значения: {unsigned char r; unsigned char g; unsigned char b}) (для P5 – массив unsigned char, т.к. всего один канал).
2. Запустим счетчик времени.
3. Проходимся по массиву pixels и собираем информацию о количестве каждого значения в канал по отдельности (записывая в массивы длиной 256, т.к. значений всего 256). Проходимся, распараллеливая цикл, при этом в каждом потоке создаем массив buffer[256] и заполняем его отдельно для каждого потока, а затем в зоне “critical” мерджим buffer и основной массив счетчиков. Так мы избегаем конфликтов в подсчете.
4. Найдем минимум/максимум в каждом канале с учетом коэффициента пропуска (k-я порядковая статистика). Для RGB т.к. канала 3, то можно запустить 3 функции find_min_max() параллельно в зоне “sections”. После этого находим минимум/максимум из всех каналов.
5. Теперь если минимум не равен 0, а максимум 255, то запускаем цикл, в котором будем изменять значения пикселей. При этом, чтобы постоянно не запускать функцию convert() (которая применяет нашу формулу), запомним все «преобразованные значения» в массив predict[256].
6. Остановим счетчик времени. Выведем результат.
7. Откроем выходной файл для записи с помощью std::ofstream. Запишем заголовок (специальные значения, например “P6/P5”, ширина, высота…)

Замеры скорости

Произведем замеры скорости программы при различных условиях. В качестве значения будем брать среднее арифметическое из времени работы 10 запусков.

Графики скорости работы программы при Schedule(static, {1, 10, 1000}):

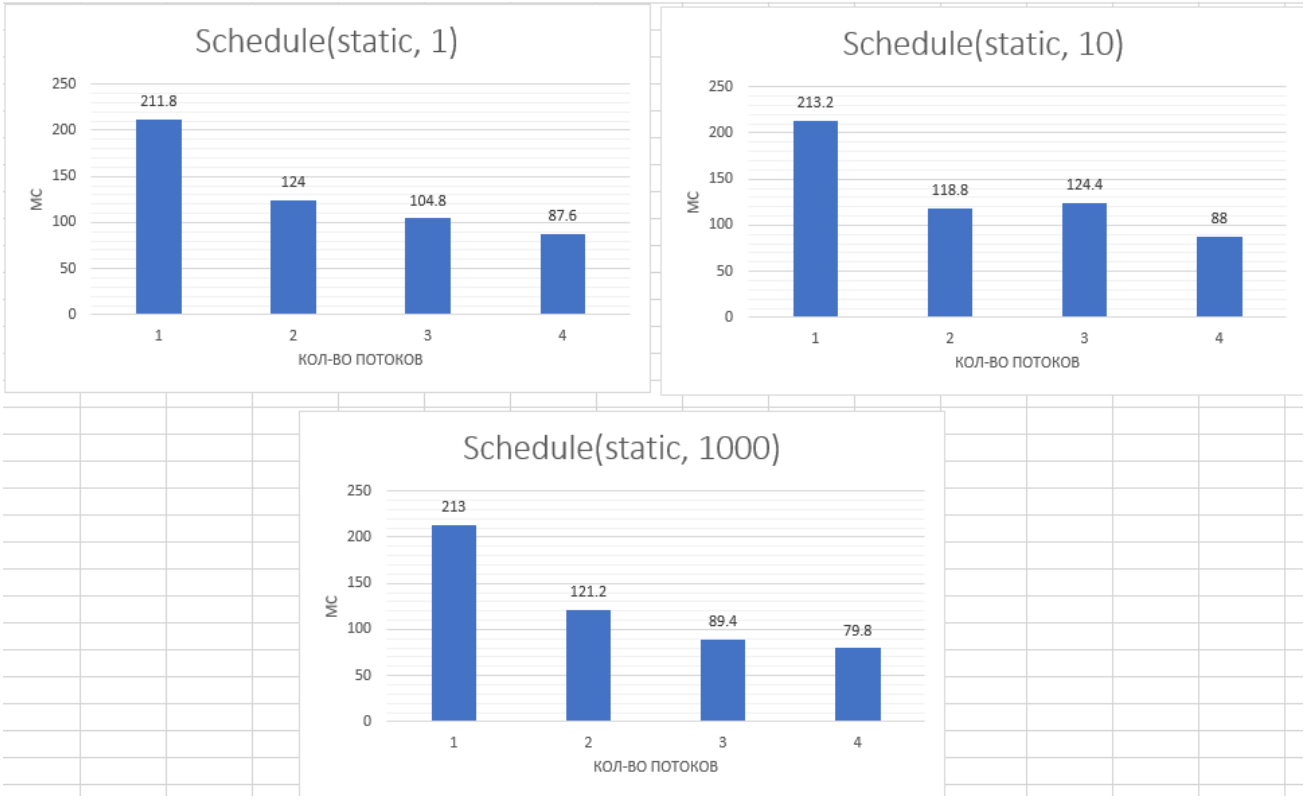

Рисунок №1 – Schedule(static, {1, 10, 1000}).

Графики скорости работы программы при Schedule(dynamic, {1, 10, 1000}):

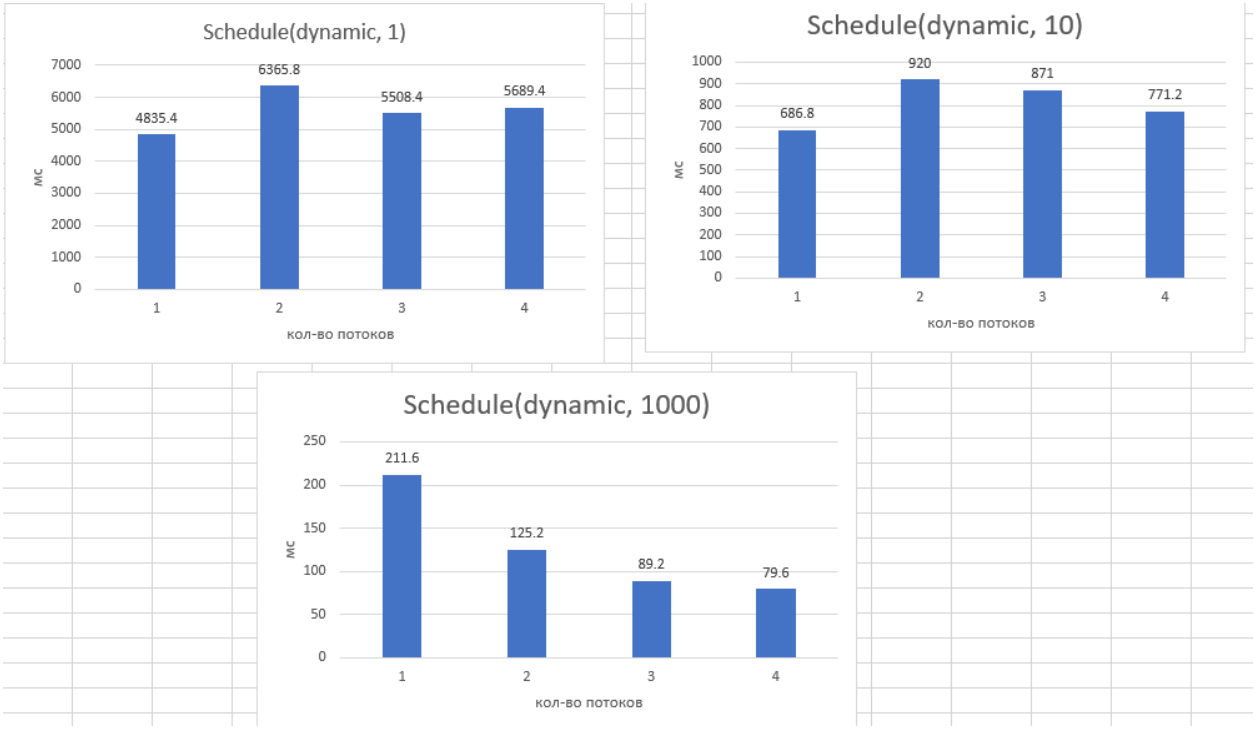

Рисунок №2 – Schedule(dynamic, {1, 10, 1000}).

Графики скорости работы программ при thread_num={1,2,3,4}.

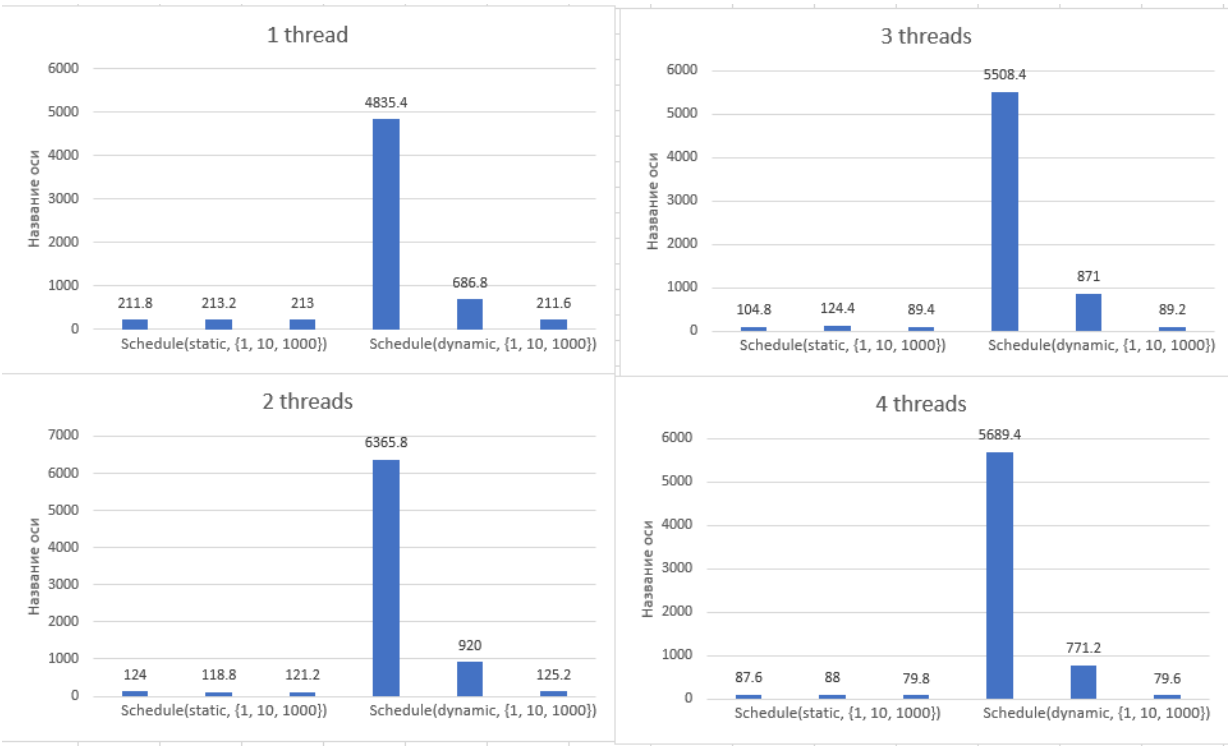

Рисунок №3 – thread_num={1,2,3,4}.

Из данных графиков следует вывод, что для данной реализации алгоритма следует использовать static, т.к. получаются более стабильные результаты.

График скорости работы программы с включенным OpenMP (1 поток) и отключенным:

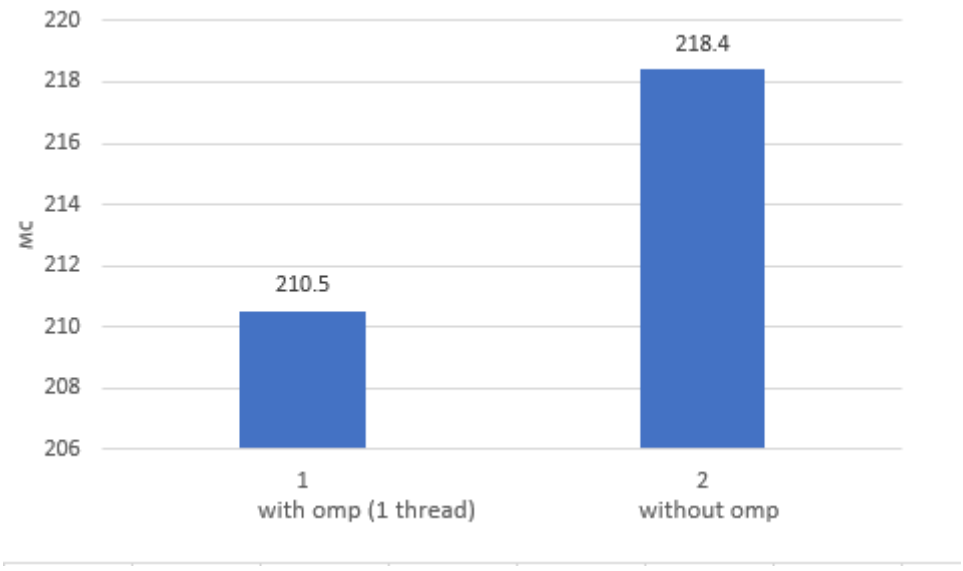

Рисунок №4 – with omp / without omp

Из данного графика видно, что программа без OpenMP не сильно медленнее, нежели программа с OpenMP (1 поток).

Листинг

```
#include <iostream>
#include <stdio.h>
#include <string>
#include <fstream>
#include <omp.h>

using namespace std;

struct pix {
    unsigned char r;
    unsigned char g;
```

```
    unsigned char b;
};

struct brightness {
    int min;
    int max;
};

unsigned char convert(float x, float max_br, float min_br) {
    if (max_br <= min_br) {
        return max_br;
    }
    int answ = ((x - min_br) / (max_br - min_br)) * 255;
    if (answ > 255) {
        return 255;
    }
    if (answ < 0) {
        return 0;
    }
    return answ;
}

int max(int a, int b) {
    if (a >= b) {
        return a;
    }
    return b;
}

int min(int a, int b) {
    if (a <= b) {
        return a;
    }
```

```
    return b;
}

void arrprint(int *arr) {
    for (int i = 0; i < 255; i++) {
        printf("<%d:%d> ", i, arr[i]);
    }
    printf("\n");
}

brightness find_min_max(int *pixels_counter, int propusk) {
    int propusk_min = propusk, propusk_max = propusk;
    int min_br, max_br;

    bool found_min = false, found_max = false;

//    arrprint(pixels_counter);

    for (int i = 0; i < 256; i++) {
        if (!found_min && pixels_counter[i] > 0) {
            if (pixels_counter[i] > propusk_min) {
                min_br = i;
                found_min = true;
                if (found_max) {
                    break;
                }
            } else {
                propusk_min -= pixels_counter[i];
            }
        }
        if (!found_max && pixels_counter[255 - i] > 0) {
            if (pixels_counter[255 - i] > propusk_max) {
                max_br = 255 - i;
```

```
                found_max = true;
                if (found_min) {
                    break;
                }
            } else {
                propusk_max -= pixels_counter[255 - i];
            }
        }
    }

    brightness answ{min_br, max_br};
    return answ;
}

int main(int argc, char *argv[])
{
    if (argc < 5) {
        printf("Not 4 args...");
        return 0;
    }
    string name_in;
    string name_out;
    float coef;
    try {
        name_in = argv[2];
        name_out = argv[3];
        coef = stof(argv[4]);

    } catch (const std::exception&error) {
        printf("Error!\n");
        return 0;
    }
    if (coef >= 0.5 || coef < 0.0) {
```

```
    printf("Incorrect coeff %f\n", coef);
    return 0;
}

std::ifstream infile(name_in, fstream::in | fstream::binary);

if (!infile.is_open()) {
    printf("File not found...");
    return 0;
}
int num_thr1;
try {
    num_thr1 = stoi(argv[1]);
} catch (const std::exception&error) {
    printf("Incorrect number of thread(s)");
    return 0;
}
if (num_thr1 < 1) {
    num_thr1 = omp_get_max_threads();
}
const int num_thr = num_thr1;
omp_set_num_threads(num_thr);
char c;
infile.get(c);
if (c == 'P') {
    infile.get(c);
    if (c == '6') {
        infile.get(c);
        infile.get(c);

        char wi[20], hi[20], maxi[20];
        int pos_wi = 0, pos_hi = 0, pos_maxi = 0;
        int w = 0;
```

```
            while (c != ' ' && c != '\n') {
                wi[pos_wi++] = c;
                w = (w * 10) + (c - '0');
                infile.get(c);
            }
            int h = 0;
            infile.get(c);
            while (c != ' ' && c != '\n') {
                hi[pos_hi++] = c;
                h = (h * 10) + (c - '0');
                infile.get(c);
            }
            int maximum = 0;
            infile.get(c);
            while (c != ' ' && c != '\n') {
                maxi[pos_maxi++] = c;
                maximum = (maximum * 10) + (c - '0');
                infile.get(c);
            }
//              printf("width: %d\nheight: %d\nmax_color(?): %d\n", w, h,
maximum);

            pix *pixels = new pix[h * w];
            int max_br = 0, min_br = 255;
            int propusk = ((float) (h * w)) * coef;
            int pixels_counter_r[256]{};
            int pixels_counter_g[256]{};
            int pixels_counter_b[256]{};

            for (int i = 0; i < h * w; i++) {
                unsigned char c1;

                infile.get(c);
                c1 = (unsigned char) c;
```

```
                pixels[i].r = c1;

                infile.get(c);
                c1 = (unsigned char) c;
                pixels[i].g = c1;

                infile.get(c);
                c1 = (unsigned char) c;
                pixels[i].b = c1;
            }
            infile.close();

            double start_time = omp_get_wtime();

#pragma omp parallel default(none) shared(h, w, pixels, pixels_counter_r,
pixels_counter_g, pixels_counter_b)
            {
                int buffer_r[256] = {0};
                int buffer_g[256] = {0};
                int buffer_b[256] = {0};
#pragma omp for schedule(static)
                for (int i = 0; i < h * w; i++) {
                    buffer_r[pixels[i].r]++;
                    buffer_g[pixels[i].g]++;
                    buffer_b[pixels[i].b]++;
                }
#pragma omp critical (merge)
                {
                    for (int j = 0; j < 256; j++) {
                        pixels_counter_r[j] += buffer_r[j];
                        pixels_counter_g[j] += buffer_g[j];
                        pixels_counter_b[j] += buffer_b[j];
                    }
                }
```

```
            }

            brightness minmax_r;
            brightness minmax_g;
            brightness minmax_b;
#pragma omp parallel sections default(none) shared(minmax_r, minmax_g, minmax_b,
pixels_counter_r, pixels_counter_g, pixels_counter_b, propusk)
            {
#pragma omp section
                minmax_r = find_min_max(pixels_counter_r, propusk);
#pragma omp section
                minmax_g = find_min_max(pixels_counter_g, propusk);
#pragma omp section
                minmax_b = find_min_max(pixels_counter_b, propusk);
            }
            min_br = minmax_r.min;
            min_br = min(min_br, min(minmax_g.min, minmax_b.min));
            max_br = minmax_r.max;
            max_br = max(max_br, max(minmax_g.max, minmax_b.max));

//            printf("min = %d\tmax = %d\n", min_br, max_br);
            if (min_br == 0 && max_br == 255) {

            } else {
                unsigned char predict[256];
                for (int i = 0; i < 256; i++) {
                    predict[i] = convert(i, max_br, min_br);
                }

#pragma omp parallel default(none) shared(pixels, predict, h, w)
                {
#pragma omp for schedule(static)
                    for (int i = 0; i < h * w; i++) {
```

```
            pixels[i].r = predict[pixels[i].r];
            pixels[i].g = predict[pixels[i].g];
            pixels[i].b = predict[pixels[i].b];
        }
    }
}

double end_time = omp_get_wtime();
double duration = end_time - start_time;

printf("Time (%i thread(s)): %g ms\n", num_thr, duration * 1000);

ofstream outfile(name_out, ios::out | ios::binary);
outfile.put('P');
outfile.put('6');
outfile.put('\n');
for (int i = 0; i < pos_wi; i++) {
    outfile.put(wi[i]);
}
outfile.put(' ');
for (int i = 0; i < pos_hi; i++) {
    outfile.put(hi[i]);
}
outfile.put('\n');
for (int i = 0; i < pos_maxi; i++) {
    outfile.put(maxi[i]);
}
outfile.put('\n');

for (int i = 0; i < w * h; i++) {
    outfile.put((unsigned char) pixels[i].r);
    outfile.put((unsigned char) pixels[i].g);
    outfile.put((unsigned char) pixels[i].b);
```

```
            }

            outfile.close();

            delete[] pixels;
        } else if (c == '5') {
            infile.get(c);
            infile.get(c);

            char wi[20], hi[20], maxi[20];
            int pos_wi = 0, pos_hi = 0, pos_maxi = 0;
            int w = 0;
            while (c != ' ' && c != '\n') {
                wi[pos_wi++] = c;
                w = (w * 10) + (c - '0');
                infile.get(c);
            }
            int h = 0;
            infile.get(c);
            while (c != ' ' && c != '\n') {
                hi[pos_hi++] = c;
                h = (h * 10) + (c - '0');
                infile.get(c);
            }
            int maximum = 0;
            infile.get(c);
            while (c != ' ' && c != '\n') {
                maxi[pos_maxi++] = c;
                maximum = (maximum * 10) + (c - '0');
                infile.get(c);
            }
//            printf("width: %d\nheight: %d\nmax_color(?): %d\n", w, h,
maximum);
```

```
            unsigned char *pixels = new unsigned char[h * w];
            int max_br = 0, min_br = 255;
            int propusk = ((float) (h * w)) * coef;
//            int *pixels_counter = new int[256]{};

            int pixels_counter[256]{};

            for (int i = 0; i < h * w; i++) {
                unsigned char c1;

                infile.get(c);
                c1 = (unsigned char) c;
                pixels[i] = c1;
            }
            infile.close();

            double start_time = omp_get_wtime();

#pragma omp parallel default(none) shared(h, w, pixels, pixels_counter)
            {
                int buffer[256] = {0};
#pragma omp for schedule(static)
                for (int i = 0; i < h * w; i++) {
                    buffer[pixels[i]]++;
                }
#pragma omp critical (merge)
                {
                    for (int j = 0; j < 256; j++) {
                        pixels_counter[j] += buffer[j];
                    }
                }

            }
```

```
            brightness minmax = find_min_max(pixels_counter, propusk);
            min_br = minmax.min;
            max_br = minmax.max;

            if (min_br == 0 && max_br == 255) {

            } else {
                unsigned char predict[256];
                for (int i = 0; i < 256; i++) {
                    predict[i] = convert(i, max_br, min_br);
                }

#pragma omp parallel for default(none) schedule(static) shared(h, w, pixels,
predict)
                for (int i = 0; i < h * w; i++) {
                    pixels[i] = predict[pixels[i]];
                }

            }

            double end_time = omp_get_wtime();
            double duration = end_time - start_time;

            printf("Time (%i thread(s)): %g ms\n", num_thr, duration * 1000);

            ofstream outfile(name_out, ios::out | ios::binary);
            outfile.put('P');
            outfile.put('5');
            outfile.put('\n');
            for (int i = 0; i < pos_wi; i++) {
                outfile.put(wi[i]);
            }
            outfile.put(' ');
```

```
            for (int i = 0; i < pos_hi; i++) {
                outfile.put(hi[i]);
            }
            outfile.put('\n');
            for (int i = 0; i < pos_maxi; i++) {
                outfile.put(maxi[i]);
            }
            outfile.put('\n');

            for (int i = 0; i < w * h; i++) {
                outfile.put((unsigned char) pixels[i]);
            }

            outfile.close();

            delete[] pixels;
        } else {
            printf("File isn't P5 or P6...");
            return 0;
        }
    } else {
        printf("File isn't P5 or P6...");
    }

//    printf("The above code block was executed in %.4f second(s)\n", ((double)
end - start) / ((double) CLOCKS_PER_SEC));


    return 0;
}
```