САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил: Пономаренко Илья Сергеевич

Номер ИСУ: 334831

студ. гр. М3135

Санкт-Петербург

2021

Цель работы: знакомство со стандартом OpenMP.

Инструментарий и требования к работе: C++, Стандарт OpenMP 2.0.

Теоретическая часть

OpenMP – библиотека для параллельного программирования. Данная библиотека используется для программирования систем с общей памятью, где потоки создаются в рамках единого процесса, и имеют как собственную память, так и доступ к общей. Схематично это представлено на рисунке 1.

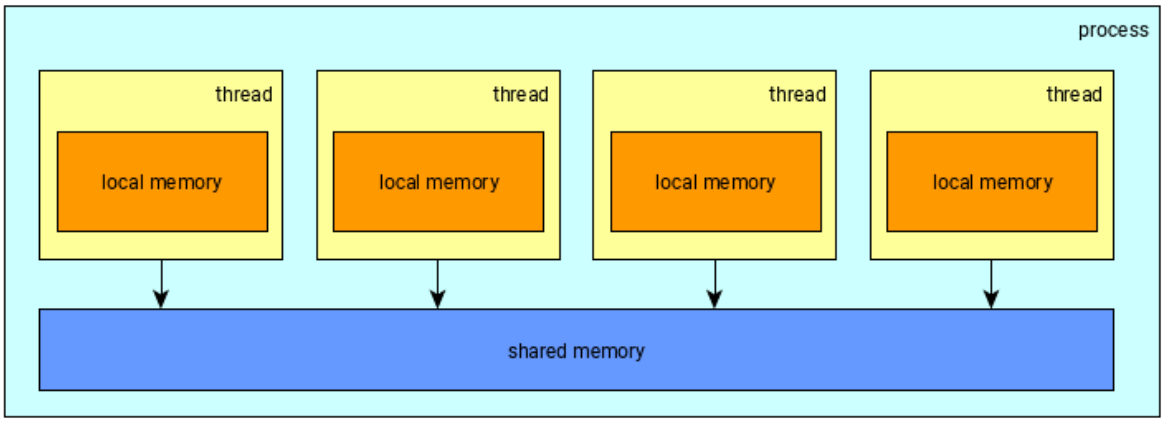


Рисунок №1 – Устройство потоков и памяти в OpenMP

Для создания потоков и распараллеливания используется #pragma omp parallel. Настроить количество потоков можно с помощью функции omp_set_num_threads(), а узнать максимальное значение потоков для текущей машины с помощью omp_get_max_threads(). Для того, чтобы распараллелить цикл for, мы можем написать #pragma omp parallel for, затем указать следующие аргументы.

shared(variable-list) – в неё мы передаём те переменные, к которым хотим иметь доступ внутри параллельной секции.

default(none|shared):

- none – не предоставлять доступ к остальным переменным внутри параллельной секции.
- shared – предоставлять доступ ко всем переменным внутри параллельной секции.

schedule(kind[, chunk_size]) – определяет то, каким образом итерации цикла будут распределены между потоками.

chunk_size – целочисленный размер блока.

kind:

- static – блоки назначаются циклическим способом в порядке номеров потоков, по умолчанию chunk_size равен количеству итераций делённых на число потоков.
- dynamic – блоки назначаются динамическим способом, в порядке ожидающих назначения, по умолчанию chunk_size равен 1.

Контраст изображения – отношение яркостей самой светлой и самой тёмной частей изображения.

По изображению можно построить гистограмму – график статистического распределения элементов цифрового изображения с различной яркостью, в котором по горизонтальной оси представлена яркость, а по вертикали относительное число пикселей с конкретным значением яркости.

Автоматическая контрастность – это метод сопоставления минимальной и максимальной яркостей, обнаруженных на изображении, с минимальной и максимальной яркостью полного диапазона соответственно (в случае восьмибитного изображения полный диапазон составляет [0; 255]).

Формула автоматической контрастности определяется как:

$$f_{\mathrm{ac}}(a) = (a - a_{\mathrm{low}}) \cdot \frac{255}{a_{\mathrm{high}} - a_{\mathrm{low}}}$$

Самая высокая и самая низкая яркости могут быть шумом изображения. Мы хотим исключить этот шум, указав коэффициент – процент пропущенных значений при подсчёте максимума и минимума.

Ниже приложил пример работы алгоритма и гистограммы к примеру (см. рис 2, 3, 4, 5).

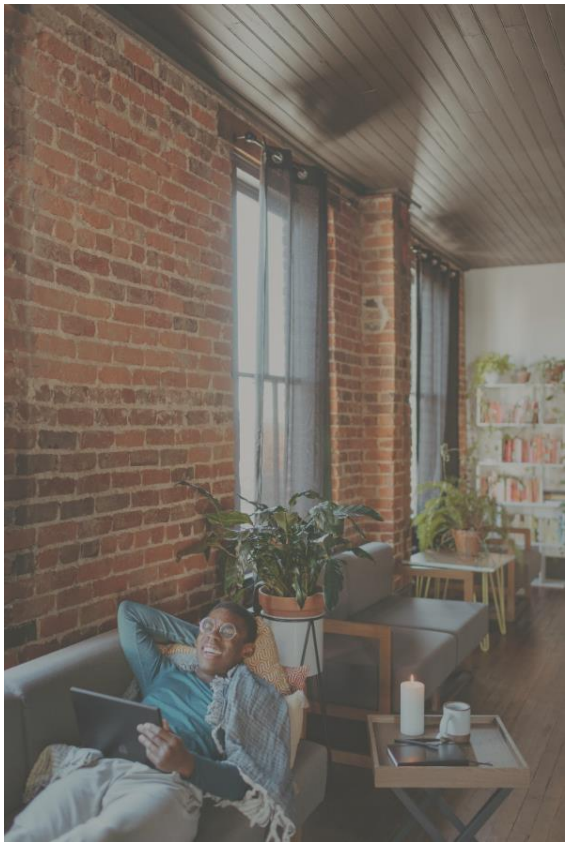


Рисунок №2 – Пример фото до обработки.

Рисунок №3 – Пример фото после обработки.

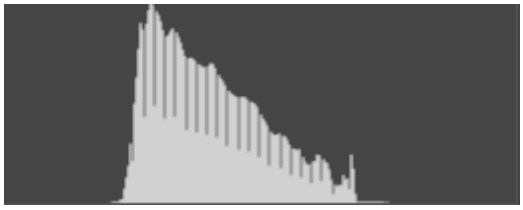


Рисунок №4 – Гистограмма до обработки.

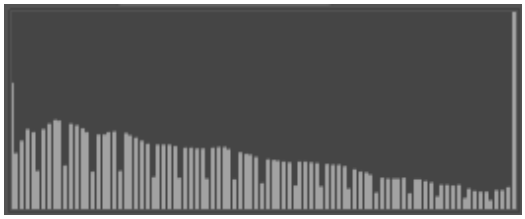


Рисунок №5 – Гистограмма после обработки.

Как мы можем заметить, гистограмма стала не такой ровной, к сожалению, это связанно с тем, что часть значений теряются при растяжении.

Практическая часть

Первым делом я считываю файл функцией fread в массив arr. Затем проверяю на соответствие формата portable bitmap. Считываю ширину, высоту, смотрю на то, является ли фото цветным. Затем засекаю время и начинаю обработку фотографии.

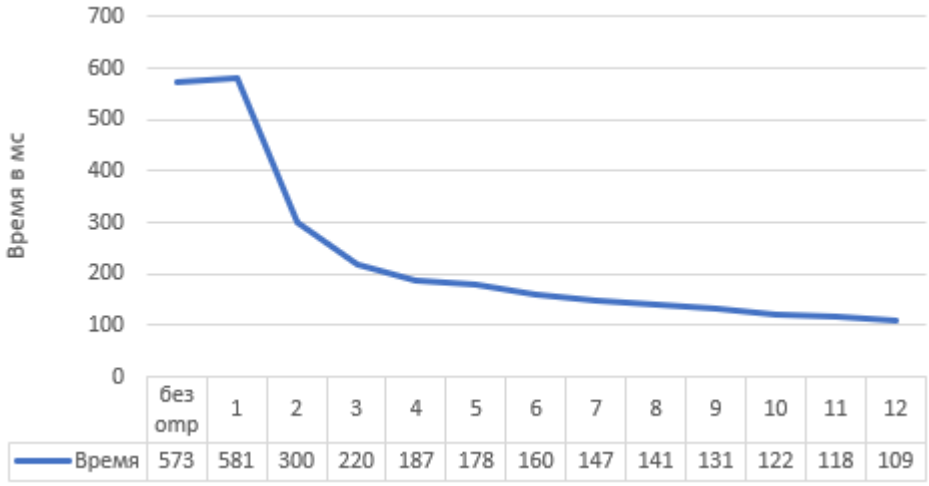
Для реализации нахождения минимума и максимума я использую подсчёт количества вхождения значений в каналы r, g, b. В дальнейшем, если фото не цветное, я использую только один канал r. Массивы r, g, b будут заполнены количеством вхождений этого значения в файле по каналам. Теперь возник вопрос, как же это можно распараллелить? Очевидно, что нативная реализация очень проста, мы можем просто пробежать по всем значениям, увеличивая значение канала по этому ключу на 1. Но нам нужна параллельность! Я нашёл такую вещь, как reduction, но,

увы, для массивов она не доступна в стандарте 2.0. Я реализовал нечто похожее на это, но своими силами. Я решил разделить наш массив на n частей, где n – количество потоков. Так же увеличив величину разреза до значения кратному трём если картинка цветная, чтобы избежать смещения относительно каналов. И для каждого потока посчитать свои значения счётчиков, а потом сделать merge с глобальными счётчиками в критической секции, чтобы избежать race condition. Мне показался этот способ самым оптимальным. А самое главное, что он ещё и работает, не уступая в скорости reduction.

Как же теперь найти минимум и максимум? Я написал функции get_mn и get_mx, чтобы находить минимум и максимум соответственно для одного из каналов. Осталось найти глобальный минимум и максимум, который мы будем подставлять в нашу формулу. Очевидно, что это минимум из минимумов по каналом, и максимум из максимумов по каналам соответственно. Затем я делаю пересчёт значений по формуле, который записываю в массив values, где по индексу (значение до) лежит значение после. Затем я распараллелил цикл изменения самих значений исходной фотографии, очевидно, что операции независимы друг от друга и мы можем это сделать просто дописав прагму.

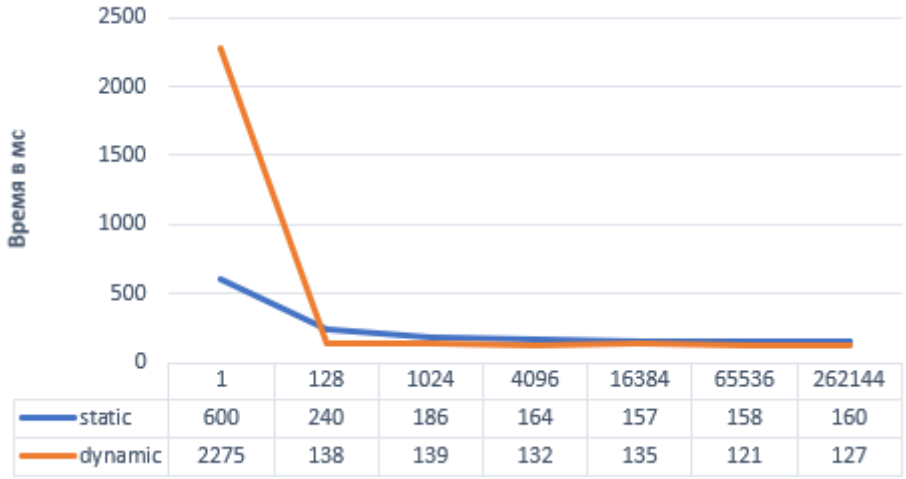
Ну и в конце я вывожу время, затраченное на обработку фотографии и произвожу запись в файл, с дальнейшим удалением массива из кучи.

Ниже представил графики зависимости времени работы от настроек omp (см. рис. 6, 7).



	без omp	1	2	3	4	5	6	7	8	9	10	11	12
Время	573	581	300	220	187	178	160	147	141	131	122	118	109

Рисунок №6 – Зависимость времени работы от числа потоков.



	1	128	1024	4096	16384	65536	262144
static	600	240	186	164	157	158	160
dynamic	2275	138	139	132	135	121	127

Рисунок №7 – Зависимость времени работы от размера блоков (12 потоков).

Листинг

Проводил тесты на компиляторах: Intel C++ 2022, g++ 9.3.0.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)
project(hw5)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "-fopenmp -O3 -mfma")

add_executable(hw5 main.cpp)
```

main.cpp

```
#include <iostream>
#include <omp.h>
#include <string>
#include <algorithm>
#include <cmath>

using namespace std;

inline unsigned char get_mn(const long *r, long need) {
    long c = 0;
    for (long i = 0; i < 256; i++) {
        c += r[i];
        if (c > need) {
            return i;
        }
    }
    return 255;
}

inline unsigned char get_mx(const long *r, long need) {
    long c = 0;
    for (long i = 255; i >= 0; i--) {
        c += r[i];
        if (c > need) {
            return i;
        }
    }
    return 0;
}

inline unsigned char overflow(double x) {
    if (x > 255) return 255;
    if (x < 0) return 0;
    return (unsigned char) round(x);
}

int main(int argc, char *argv[]) {

    if (argc < 5) {
```

```
        cout << "invalid arguments" << endl;
        return 0;
    }

    int thr;
    double threshold;

    try {
        thr = stoi(argv[1]);
        threshold = stof(argv[4]);
    } catch (const invalid_argument &e) {
        cout << "Error in " << e.what() << endl << "It is not a number" <<
endl;
        return 0;
    }

    thr = thr == 0 ? omp_get_max_threads() : thr;
    omp_set_num_threads(thr);

    FILE *input = fopen(argv[2], "rb");
    if (!input) {
        cout << "Wrong file name" << endl;
        return 0;
    }
    fseek(input, 0, SEEK_END);
    long size = ftell(input);
    fseek(input, 0, SEEK_SET);
    auto *arr = new unsigned char[size];
    if (size != fread(arr, sizeof(unsigned char), size, input)) {
        cout << "Something went wrong while read, try again." << endl;
        return 0;
    }
    fclose(input);

    if (arr[0] != 'P' || (arr[1] != '5' && arr[1] != '6')) {
        cout << "It is not portable bitmap image file" << endl;
        return 0;
    }
    bool rgb = arr[1] == '6';
    long w = 0, h = 0, pos = 2;
    while (isspace(arr[pos])) pos++;
    pos--;
    while (!isspace(arr[++pos])) {
        w *= 10;
        w += arr[pos] - '0';
    }
    while (!isspace(arr[++pos])) {
        h *= 10;
        h += arr[pos] - '0';
    }
    while (isspace(arr[pos++]));
    pos += 3;

    double start = omp_get_wtime();

    long r[256] = {0};
    long g[256] = {0};
    long b[256] = {0};
```

```
    int step = rgb ? 3 : 1;

    long block = size / thr;
    while (rgb && block % 3) block++;
#pragma omp parallel for shared(arr, block, pos, size, rgb, step, r, g, b,
cout) default(none) schedule(dynamic)
    for (long k = pos; k < size; k += block) {
        long rc[256] = {0}, gc[256] = {0}, bc[256] = {0};
        long end = min((long) size, k + block);
        for (long i = k; i < end; i += step) {
            rc[arr[i]]++;
            if (rgb) {
                gc[arr[i + 1]]++;
                bc[arr[i + 2]]++;
            }
        }
#pragma omp critical
        {
            for (int i = 0; i < 256; i++) {
                r[i] += rc[i];
                g[i] += gc[i];
                b[i] += bc[i];
            }
        }
    }

    long need = (long) (w * h * threshold);
    unsigned char mn = rgb ? min({get_mn(r, need), get_mn(g, need), get_mn(b,
need)}) : get_mn(r, need);
    unsigned char mx = rgb ? max({get_mx(r, need), get_mx(g, need), get_mx(b,
need)}) : get_mx(r, need);

    unsigned char values[256];
    for (int i = 0; i < 256; i++) {
        values[i] = overflow(255.0 * ((double) (i - mn) / (double) (mx -
mn)));
    }

    if (mn != mx) {
#pragma omp parallel for shared(arr, pos, size, values) default(none)
schedule(dynamic, 65536)
        for (long i = pos; i < size; i++) {
            arr[i] = values[arr[i]];
        }
    }

    cout << "Time (" << thr << " thread(s)): " << 1000 * (omp_get_wtime() -
start) << " ms" << endl;
    FILE *output = fopen(argv[3], "wb");
    fwrite(arr, sizeof(unsigned char), size, output);
    fclose(output);
    delete[] arr;
}
```