Лабораторная работа №4	M3136	2022
OpenMP	Тарасевич Артем Сергеевич	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: язык C++20, компилятор g++ (GCC) 12.2.0. Стандарт OpenMP 2.0.

Конструкции OpenMP

Библиотека OpenMP в языке C++ используется в виде вызовов функций и использования директив `#pragma omp` (сделано это в угоду безопасности, чтобы при использовании других библиотек, использующих директиву `#pragma`, не происходило конфликтов). Полный формат использования выглядит следующим образом:

```
#pragma omp <название директивы> [параметр-1, параметр-2...]
```

Так как `#pragma` обрабатывается препроцессором (как и любая другая команда, начинающаяся с решетки) то каждая такая директива не должна содержать лишних символов в конце. Поэтому, также нельзя использовать несколько директив на одной строчке (исключения присутствуют, но их немного).

Конструкция `parallel`

```
#pragma omp parallel <параметр1, параметр2...>
{
    ...
}
```

Ключевая конструкция - используется для обозначения “параллельного региона” (область кода, которая исполняется несколькими потоками). Параметрами данной конструкции могут быть:

1. `if (clause)` - если значение clause равно нулю, то данный “регион” перестает быть параллельным и запускается в однопоточном формате
2. `private\shared\firstprivate\lastprivate (variables)` - определяет видимость переменных variables для каждого потока, где private - отдельные переменные для каждого потока, a shared - общие.
3. `default (shared | none)` - определяет стандартную видимость. Если установлено значение none, то видимость переменных, которые фигурировали до конструкции “parallel” и используются в этой конструкции, должны быть прописаны явно.
4. `num_threads (expression)` - определяет, сколько потоков должно исполнять параллельный регион. Имеет больший приоритет, чем значение `OMP_NUM_THREADS` и функция `omp_set_num_threads`.

Поведение последнего параметра также определяет функция `void omp_set_dynamic(int)`, которая отвечает за динамическое регулирование числа потоков (dynamic adjustment). Так как по-умолчанию регулирование динамическое, то параметр `num_threads` играет роль максимального количества выделяемых потоков, потому потом потребуется прописать `omp_set_dynamic(0)`, чтобы выключить регулирование.

В конце каждого “параллельного региона” стоит неявный барьер (см. конструкцию `barrier`).

Конструкция `for`

```
#pragma omp for <параметр1, параметр2…>
     for (init; var logical-op b; incr)
```

Версия цикла с совмещенным блоком `parallel`:

```
#pragma omp parallel for <параметр1, параметр2…>
     for (init; var logical-op b; incr)
```

Используется для распараллеливания циклов. Данная директива накладывает ограничения на вид цикла: в части `init` можно только инициализировать одну переменную, на месте `b` константа (или функция,

значение которой постоянно), на месте `var` стоит та же переменная, что и инициализировалась, логическим оператором `logical-op` могут быть операторы сравнения (`<, <=, >, >=`), а в incr может быть только инкремент переменной. К тому же, цикл не должен завершаться операцией `break`.

Ключевым параметром в данной конструкции является `schedule(kind, chunk_size)` - способ разбиения итераций цикла в “параллельных регионах”. Выделяется четыре типа:

1. `static` - разбиение происходит на куски (кусок - последовательные итерации). Данные куски распределяются в формате round-robin, т.е. отдаются по кругу. Если значение chunk_size не установлено, то оно подбирается так, чтобы каждому потоку достался ровно один кусок.
2. `dynamic` - разбиение также происходит на куске, но распределение происходит “по-факту”: когда происходит выбор, кому отдать кусок, то кусок отдается свободному потоку. Если значение chunk_size не указано, то оно выбирается равным единице.
3. `guided` - куски в этом формате разные по размеру: начальные куски большие, а к концу они становятся все меньше и меньше.
4. `runtime` - используется для того, чтобы выставить в режиме работы программы (runtime) тип разбиения цикла.

К прочему, параметрами могут также являться

1. `ordered` - итерации цикла должны выполняться последовательно друг за другом
2. `nowait` - удаление неявного `barrier` в конце цикла.

Конструкция `critical`

```
#pragma omp critical
{
   …
}
```

Данная конструкция определяет блок, который выполняется только одним потоком. Если несколько потоков входит в данный блок, то выбирается один из них и выполняет блок, а остальные покорно ждут свой черед.

Конструкция `barrier`

```
#pragma omp barrier
```

Данная конструкция используется для синхронизации позиций потоков. Если потоки были каким-то образом разъединены (например конструкцией `nowait`), то данная директива их синхронизирует, т. е. дальнейшее выполнение команд начнется “одновременно”.

Описание работы кода

Программа разбита на 5 файлов:

1. readData.cpp - файл, отвечающий за считывание картинки и проверки его характеристик (формат P5, количество яркостей)
2. writeData.cpp - файл, отвечающий за запись картинки
3. image.cpp - файл, в котором находятся функции:
    a. `getHistogramm` - подсчет гистограммы. Возвращает массив из 256 целых чисел, где отдельная ячейка отвечает за встречаемость пикселя данной интенсивности.
    b. `getFrequenciesOfPixels` - перевод из гистограммы в вероятности отдельных пикселей. Возвращает массив.
    c. `getFilteredImage` - функция, которая, в соответствии с порогами, изменяет исходное изображение.
    d. `getBestThresholds` - функция, занимающаяся перебором всех порогов.
4. otsuMethod - хедер, содержащий константы + типы.

Разберем данные функции по-подробнее (будет немного псевдокода для лучшей читаемости):

1. `getHistogramm`

```
array<256> getHistogramm(image, threads_count) {
     array<256> colors{};
     omp_set_dynamic(0);
       #pragma omp parallel …
         {
             array<256> colors_thread{};
             #pragma omp for nowait schedule(kind, chunk_size)
```

```
            for (int i = 0; i < img.size(); i++)
                colors_thread[img[i]]++;
            #pragma omp critical
            {
                for (int i = 0; i < 256; i++)
                    colors[i] += colors_thread[i];
            }
        }
        return colors;
    }
```

Здесь происходит распараллеливание исходной картинки следующим образом: каждому потоку отдаются какой-то куски картинки, он считывает “местную” гистограмму (т. е. гистограмму этих кусков) в секции `critical`, чтобы избежать гонки данных. Затем, значение каждого потока суммируется в общий массив.

Если пытаться считать общую гистограмму (например, используя атомарные операции) в цикле по изображению, то происходит **false-sharing -** говоря простыми словами, одна область памяти “прыгает” между кэшами, заметно замедляя программу.

2. `getFrequenciesOfPixels` устроена тривиально: происходит проход по гистограмме и деление каждого значения на общий размер картинки. Пытаться параллелить здесь бессмысленно из-за малого размера этих массивов - 256 (или может быть вовсе глупо: на создании\удалении потоков + решении проблемы false-sharing с критическими зонами можно потерять больше времени, чем в однопоточном варианте реализации)
3. `getFilteredImage` также тривиальна: происходит параллельный проход по всем пикселям и запись новых в другое изображение
4. `getBestThresholds.` Основная функция, которая полностью перебирает три порога. Выбор лучших происходит по методу Оцу: для конкретных областей яркости (областью яркости далее будем называть те цвета пикселей, что лежат меж порогов, слева исключительно, справа включительно) считается средняя яркость области μ и вероятность попадания случайного пикселя в эту область яркости ω. Оцу показал, что лучшее разбиение на области

яркости достигается при максимизации суммы $\omega_0\mu_0^2 + \omega_1\mu_1^2 + \omega_2\mu_2^2 = \sigma$.

Что касается распараллеливания, подход был применен тот же, что и при подсчете гистограммы: каждый поток занимался своей порцией порогов, находил среди своих порций лучший вариант, а затем сравнивался с другими потоками.

Результат работы программы

Картинка 500x500 Lenna:

```
Time (8 thread(s)): 4.56514 ms
77 130 187
```

Картинка 1280x720:

```
Time (8 thread(s)): 6.17477 ms
64 132 212
```

Картинка 3840x2160:

```
Time (8 thread(s)): 11.8958 ms
72 128 190
```

Процессор: AMD Ryzen 5 3500U

Экспериментальная часть

Так как в коде встречается три “параллельных региона”, и в зависимости от параметров `schedule, chunk_size`, какие-то регионы при ускоряются, а другие при тех же параметрах замедляются, то найдем сначала лучшие параметры `schedule` и `chunk_size` для отдельно взятых функций (количество потоков - 8). Чтобы как можно сильнее избавиться от погрешности замера, **использовалась** картинка с разрешением **3840x2160**.

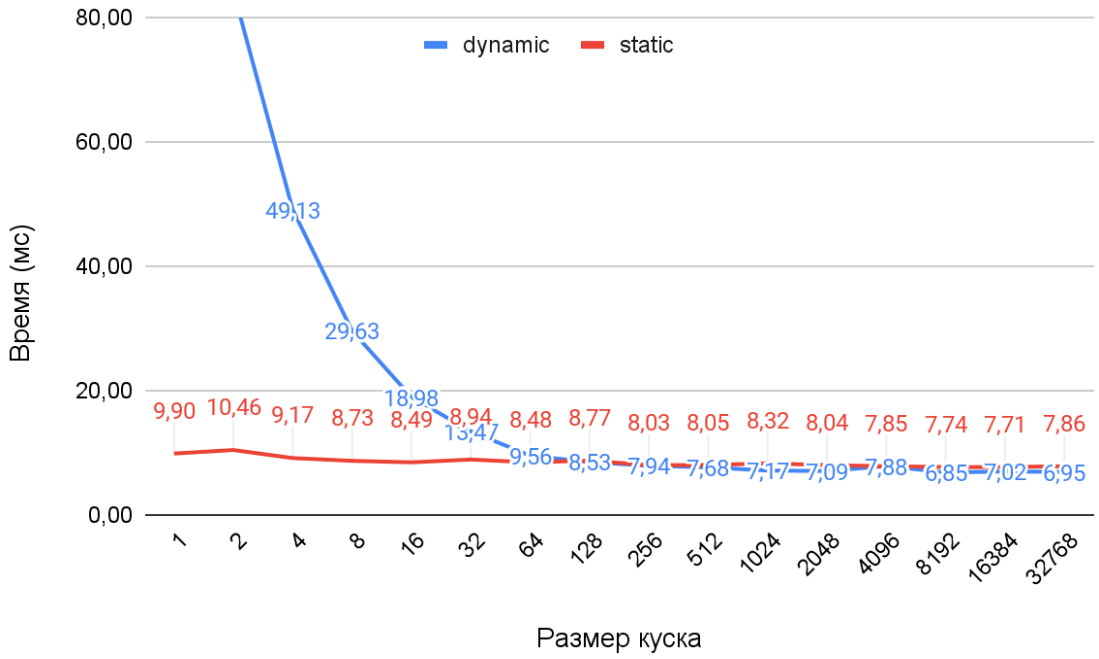

Рис 1 – Зависимость скорости пороговой фильтрации изображения от размера куска

Как видно, исходя из графика, статическое распределение почти постоянно, а динамическое убывает экспоненциально, и только к большому количеству кусков динамика начинает очень незначительно преобладать над статическим. В итоговом коде используется динамическое распределение с размером куска 32768.

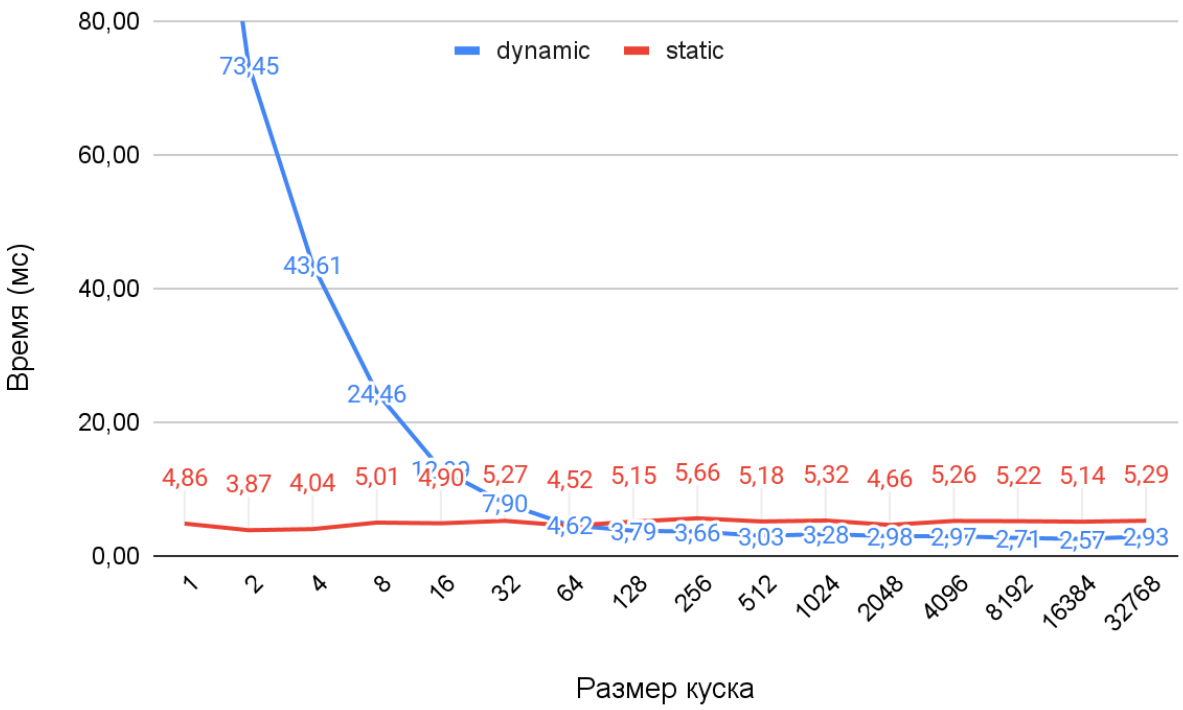

Рис 2 – Зависимость скорости подсчёта гистограммы изображения от размера куска

Здесь прослеживается та же тенденция, что и с подсчетом порогов. Потому и выбор аналогичный - динамическое распределение при размере куска 32768.

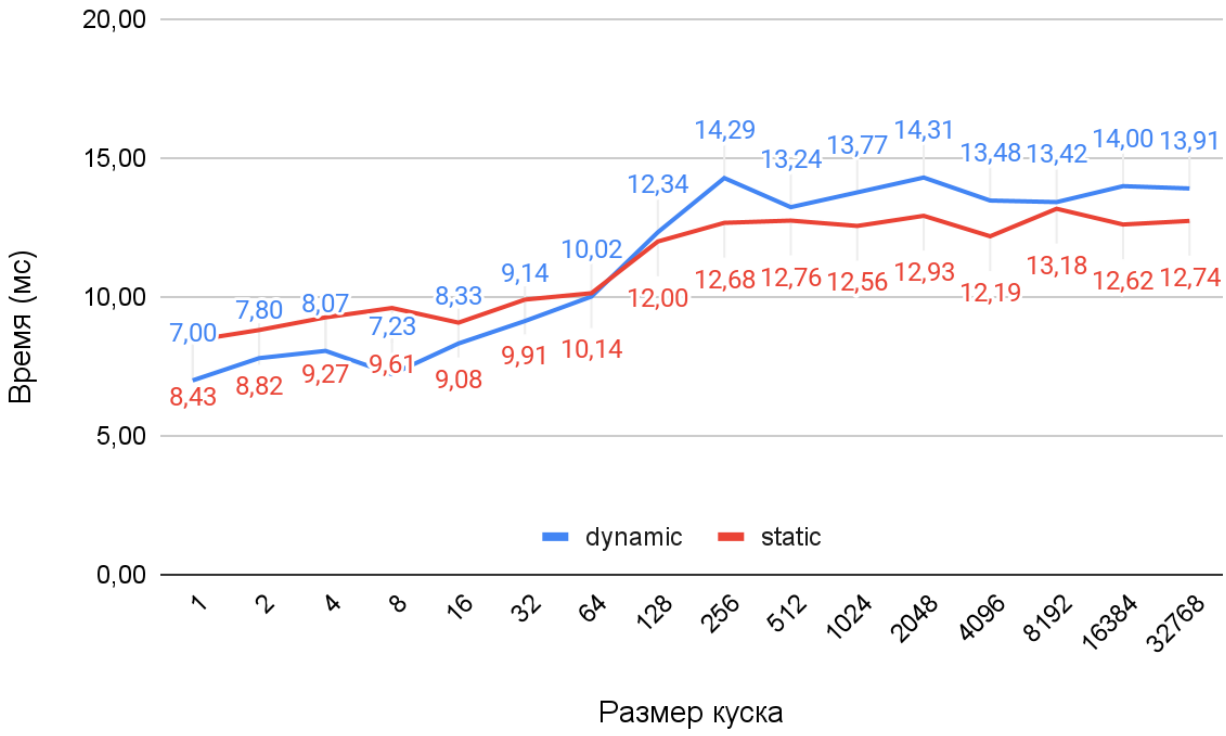

Рис 3 – Зависимость скорости алгоритма Оцу от размера куска

Так как в исходном алгоритме есть три вложенных цикла, и они не прямоугольные, а квадратные (перебор второго цикла начинается со значения итератора первого; то же самое и с третьим циклом), то все итерации по времени разные, в отличие от предыдущих двух “параллельных регионов”. Динамическое распределение с размером куска, равным единице, как видно на рисунке 3 – лучший выбор.

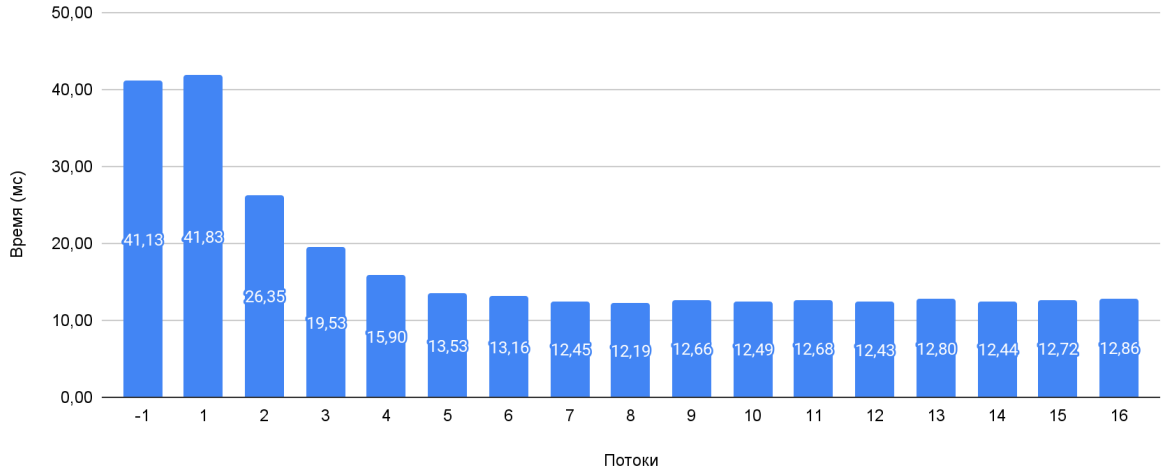

Рис 4 – Зависимость скорости работы программы от количества потоков

На рисунке 4 значение -1 отвечает за выключение многопоточности. Отсюда следует, что разницу между выключенным OpenMP и включенным 1 потоком можно сослать на погрешность. Как видно, оптимальное значение количества потоков от 5 и более, потом разница несущественна.

Список источников

Википедия, Метод Оцу https://en.wikipedia.org/wiki/Otsu%27s_method

Документация OpenMP https://www.openmp.org/wp-content/uploads/cspec20.pdf

Листинг кода

hard.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <array>


#include "otsuMethod.hpp"

int main(int argc, char * argv[]) {
    if (argc != 4) {
        std::cerr << "Invalid count of arguments: " << argc - 1 <<
"\nExpected: 3 (count of threads, input file, output file)";
        return -1;
    }
```

```
    int threads = -1;
    try {
        threads = std::stoi(std::string(argv[1]));
    } catch (std::exception & error) {
        std::cerr << "Couldn't parse '" + std::string(argv[1]) +
"' as number.";
        return -1;
    }

    if (threads == 0)   threads = omp_get_max_threads();

    int height, width;
    image_type image;

    try {
        const auto [h, w, im] = read_pgm(argv[2]);
        height = h; width = w; image = image_type(std::move(im));
    } catch (std::exception & error) {
        std::cerr << "Exception happened while reading file: " <<
error.what() << '\n';
        return -1;
    }

    double startTime = omp_get_wtime();
    const auto [f0, f1, f2] = getBestThresholds(image, threads);
    image_type output = getFilteredImage(image, f0, f1, f2,
threads);
    double stopTime = omp_get_wtime();

    std::printf("Time (%i thread(s)): %g ms\n", (threads == -1 ? 1
: threads), (stopTime - startTime) * 1000);

    writeImage(height, width, output, argv[3]);

    std::printf("%u %u %u\n", f0, f1, f2);
}
```

writeData.cpp

```
#include <fstream>

#include "otsuMethod.hpp"

void writeImage(int height, int width, image_type const & image,
const char * filename) {
    std::ofstream file(filename, std::ios::binary);

    if (!file.good()) {
        throw std::runtime_error("Couldn't open a file for
writing");
```

```
    }

    file << "P5\n" << height << ' ' << width << "\n255" <<
std::endl;
    for (auto const pixel: image) {
        file << pixel;
    }
}
```

image.cpp

```
#include <algorithm>
#include <iostream>
#include <array>
#include <numeric>

#include "otsuMethod.hpp"

std::array<uint64_t, 256> getHistogramm(image_type const & img,
int threads_count) noexcept {
    std::array<uint64_t, 256> colors{};

    omp_set_dynamic(0);
    #pragma omp parallel if (threads_count != -1)
num_threads(threads_count)
    {
        std::array<uint64_t, 256> colors_thread{};
        #pragma omp for nowait schedule(hs_kind, hs_chunk_size)
        for (int i = 0; i < img.size(); i++)
        {
            colors_thread[img[i]]++;
        }
        #pragma omp critical
        {
            for (int i = 0; i < 256; i++)
                colors[i] += colors_thread[i];
        }
    }
    return colors;
}

std::array<double, 256> getFrequenciesOfPixels(image_type const &
img, const int threads_count) noexcept {
    auto hs = getHistogramm(img, threads_count); int size =
img.size();
    std::array<double, 256> result{};

    for (int i = 0; i < 256; i++)
        result[i] = (double) hs[i] / size;
    return result;
```

```
}

std::tuple<int, int, int> getBestThresholds(image_type const &
img, int threads_count) noexcept {
    std::tuple<int, int, int> b_thresholds{};
    double b_sigma = -1; int size = img.size();

    std::array<double, 256> frequencies =
getFrequenciesOfPixels(img, threads_count);
    std::array<double, 256> frequenciesPrefixSums, mues;
    frequenciesPrefixSums[0] = frequencies[0]; mues[0] = 0;
    for (int i = 1; i < 256; i++) {
        frequenciesPrefixSums[i] = frequenciesPrefixSums[i - 1] +
frequencies[i];
        mues[i] = mues[i - 1] + i * frequencies[i];
    }


    omp_set_dynamic(0);
    #pragma omp parallel if (threads_count != -1)
num_threads(threads_count) shared(frequenciesPrefixSums, mues)
    {
        double b_sigma_local = -1;
        std::tuple<int, int, int> b_thresholds_local{};
        #pragma omp for schedule(otsu_kind, otsu_chunk_size)
nowait
        for (int f0 = 0; f0 <= 253; f0++) {
            for (int f1 = f0 + 1; f1 <= 254; f1++) {
                for (int f2 = f1 + 1; f2 <= 255; f2++) {
                    double
                        omega1 = frequenciesPrefixSums[f0],
                        omega2 = frequenciesPrefixSums[f1] -
frequenciesPrefixSums[f0],
                        omega3 = frequenciesPrefixSums[f2] -
frequenciesPrefixSums[f1],
                        omega4 = frequenciesPrefixSums[255] -
frequenciesPrefixSums[f2],
                        mu1 = mues[f0],
                        mu2 = mues[f1] - mues[f0],
                        mu3 = mues[f2] - mues[f1],
                        mu4 = mues[255] - mues[f2];

                    double cur_sigma = mu1 * mu1 / omega1 + mu2 *
mu2 / omega2 + mu3 * mu3 / omega3 + mu4 * mu4 / omega4;

                    if (b_sigma_local < cur_sigma) {
                        b_sigma_local = cur_sigma;
                        b_thresholds_local = std::make_tuple(f0,
```

```
f1, f2);
                    }
                }
            }
        }
        if (b_sigma < b_sigma_local) {
            #pragma omp critical
            {
                if (b_sigma < b_sigma_local) {
                    b_sigma = b_sigma_local;
                    b_thresholds = b_thresholds_local;
                }
            }
        }
    }

    return b_thresholds;
}

image_type getFilteredImage(image_type const & image, const int
f0, const int f1, const int f2, const int threads_count) {
    int size = image.size();
    image_type filteredImage(size);

    omp_set_dynamic(0);
    #pragma omp parallel for if (threads_count != -1)
num_threads(threads_count) schedule(filter_kind,
filter_chunk_size)
    for (int i = 0; i < size; i++) {
        uint8_t out;
        if      (image[i] <= f0) out = color0;
        else if (image[i] <= f1) out = color1;
        else if (image[i] <= f2) out = color2;
        else                     out = color3;
        filteredImage[i] = out;
    }
    return filteredImage;
}
```

readData.cpp

```
#include <iostream>
#include <fstream>
#include <exception>
#include <vector>
#include <tuple>
#include <sstream>
#include <string>

#include "otsuMethod.hpp"
```

```
std::tuple<int, int, image_type> read_pgm(const char * filename) {
    std::ifstream file(filename);
    if (!file.good()) {
        throw std::runtime_error("Couldn't open file");
    }

    std::string p5;
    std::getline(file, p5);

    if (p5 != "P5") {
        throw std::runtime_error("Bad input file\n: First line:
\"" + p5 + "\"; Expected: \"P5\"");
    }

    int height, width; file >> height >> width;
    if (!file.good()) {
        throw std::runtime_error("Couldn't read width & height of
image");
    }

    int mustbe255; file >> mustbe255;
    if (!file.good() || mustbe255 != 255) {
        throw std::runtime_error("Third string must be 255");
    }

    image_type image(width * height, 0);
    std::getline(file, p5);

    for (int i = 0; i < width * height; i++) {
        char ch; file.read(&ch, 1);
        image[i] = ch;
    }

    if (!file.good()) { throw std::runtime_error("IOException
while reading pixels"); }

    file.peek(); if (!file.eof()) { throw
std::runtime_error("There's some unwanted info after reading."); }
    file.close();


    return std::make_tuple(height, width, std::move(image));
}
```

otsuMethod.hpp

```
#pragma once
#include <vector>
```

```
#include <cstdint>
#include <tuple>
#include "omp.h"

using image_type = std::vector<uint8_t>;


std::tuple<int, int, image_type> read_pgm(const char * filename);
std::tuple<int, int, int> getBestThresholds(image_type const &
img, int threads_count) noexcept;
void writeImage(int height, int width, image_type const & image,
const char * filename);
image_type getFilteredImage(image_type const & image, const int
f0, const int f1, const int f2, const int threads);

std::array<uint64_t, 256> getHistogramm(image_type const & img,
int threads_count) noexcept;

const uint8_t color0 = 0, color1 = 84, color2 = 170, color3 = 255;

#define hs_kind dynamic
#define hs_chunk_size 32768
#define otsu_kind dynamic
#define otsu_chunk_size 1
#define filter_kind dynamic
#define filter_chunk_size 32768
```