Лабораторная работа №4	M3136	2022
OpenMP	Исаев Максим Викторович	

Цель работы: знакомство с основами многопоточного программирования.

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

Компилятор	g++ 12.2.0 MinGW-W64 x86_64-msvcrt-posix-seh

Описание конструкций OpenMP для распараллеливания команд.

В OpenMP все конструкции имеют следующий вид:

```
#pragma omp <directive>
{
    <code>
}
```

Pragma omp parallel: данная директива определяет регион, код в котором будет исполняться параллельно в несколько потоков. Также у этой директивы можно указать дополнительные параметры, например:

- if - позволяет включать/выключать многопоточное выполнения кода в runtime, в зависимости от значения выражения.
- default - определяет поведение по умолчанию, внутри параллельного региона для внешних переменных. (shared | none)
- shared - принимает список внешних переменных, которые являются общими для всех потоков. Если не указывать default, то все внешние переменные будут считаться shared.

- private - принимает список внешних переменных, которые должны быть свои у каждого потока. Так же все переменные объявленные внутри параллельного региона ведут себя как private.

Pragma omp for: эта директива применяется для исполнения цикла for параллельно несколькими потоками. Для её применения требуется чтобы у цикла был канонический вид, а именно:

- for (init-expr; var logical-op b; incr-expr)
- init-expr - инициализация переменной
- logical-op - логические операции: <, <=, > и >=
- incr-expr - инкремент/декремент переменной на константу.

При помощи параметра schedule можно повлиять на логику распределения итераций между потоками. schedule(type, chunk_size)

Первый аргумент тип определяет способ разбиения итераций:

- static - разбивает итерации между потоками на примерно одинаковые отрезки (iter_size/chunk_size). При этом отображение отрезков на потоки всегда одно и тоже. Если отрезков k больше чем потоков n (k > n), то n + 1 отрезок достанется первому потоку, когда тот закончит выполнять 1, n + 2 второму потоку и тд.
- dynamic - также разбивает итерации на равные отрезки, однако нет строгой привязки отрезка к потоку. Вместо этого распределение происходит по принципу “first come, first served”. (что можно перевести на русский как “кто первый встал, того и тапки”).

Второй аргумент же определяет количество итераций в отрезке. Иллюстрация с сайта http://jakascorner.com/:

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

Pragma omp critical: критическая секция - базовый примитив синхронизации. Критическая секция гарантирует, что код в ней исполняется не более чем одним потоком в каждый момент времени. Например это необходимо, когда нескольким потокам необходимо производить запись в общий ресурс. В таком случае запись производиться в критической секции.

Pragma omp atomic: аналогично критический секции гарантирует выполнение не более чем одним потоком, однако поддерживает только узкий круг “атомарных” (греч. ἄτομος — неделимых) операций (расположение памяти обновляется атомарным образом). Под это определения подходят инкремент/декремент и выражения вида:

x <binop>= Expr, где binop это один операторов:

+, *, / , - , & , << , ^ , | , >> (обязательно не перегруженный)

Часто доступны аппаратные инструкции, которые могут выполнять атомарное обновление с минимальными затратами.

Важно помнить, что критическая секция, что атомарная операция являются дорогими, так как фактически заставляют параллельный код работать последовательно.

Вариант

Я выбрал вариант Hard, в котором было необходимо реализовать алгоритм фильтрации методом Оцу (Otsu method), позволяющий разбить grayscale изображение на n классов. Алгоритм построен на основе математического доказательства импликации: из минимальной дисперсией внутри классов достижима тогда и только тогда, когда достигается максимальная дисперсия между классами. Поиск необходимого уровня осуществляется полным перебором всех возможных наборов порогов (threshold). После все оттенки между двумя порогами заменяются на один оттенок.

Стоит отметить что данный алгоритм никак не учитывает взаимного расположения пикселей, а основывается только на их встречаемости.

Например dithering позволяет добиться более визуально похожего результата, используя то же число оттенков, однако из-за того, что в этом алгоритме значение следующего пикселя вычисляется как функция от предыдущий, то такой алгоритм очень плохо параллелится. В отличие от этого итерации полного перебора в алгоритме Оцу полностью независимы друг от друга, благодаря чему и удается достичь ускорения при перебирании параллельно.

Описание работы написанного кода.

Написанный код представляет из себя класс, инкапсулирующий логику хранения, чтения и записи изображений в формате PNM, и множество функций для вычисления оптимальных пороговых значений.

Принципы выделения private и shared переменных.

Принцип разделения переменных на private и shared следующий: если поток будет много писать в переменную, то по возможности нужно сделать её private (во избежании инвалидации этой кэш-линии у других потоков), если же переменная используется только для чтения или используется за пределами параллельной зоны то она должна быть shared. Ещё раз упомяну, что любая запись в shared переменную должна производится только одним поток в момент времени, т.е необходимо использование critical или atomic section.

Параллелизм при переборе порогов.

Для подсчёта оптимальных пороговых значений проводится полный перебор по всем возможным комбинациям порогов. Это является первой зоной, которая хорошо подвержена параллелизму. 3 вложенных цикла перебирают все возможные тройки пороговых значений. При этом внешний цикл исполняется параллельно. Так как для разных итераций внешнего цикла существует разное кол-во различных значений внутренних (т.к пороги идут по возрастанию) то в этом месте целесообразно использовать dynamic-type. Не будет проблемы, что какой-то тред взял самые большие куски и тормозит всех остальных, за это время другие успеют посчитать по несколько более маленьких интервалов.

В предыдущем случае каждому потоку заводятся свои копии переменных: текущая наибольшая дисперсия между классами и массив соответствующих порогов. После параллельной зоны, лучшие результаты полученные разными тредами сравниваются и выбирается лучший из лучших. Это возможно благодаря ассоциативности функции максимум.

Параллелизм при подсчёте диаграммы.

Но для вычисления дисперсии, сначала необходимо иметь диаграмму уровней. Так как вычисление диаграммы заключается в простом подсчитывании пикселей каждого цвета (а суммирование тоже ассоциативно), то эту часть также без труда можно исполнять параллельно.

Каждому треду создаётся по массиву в которым будет хранится число встретившихся пикселей данного цвета. В конце значения из локальных массивов “сливаются” в общий.

Интересное наблюдение.

В ходе написания программы было обнаружена интересная деталь:

```
#pragma omp for
for (int y = 0; y < image.getYSize(); ++y)
{
  for (int x = 0; x < image.getXSize(); ++x)
  {
  ++localProbability[image.getPixel(x, y)];
  }
}
```

```
#pragma omp for
for (int x = 0; x < image.getXSize(); ++x)
{
  for (int y = 0; y < image.getYSize(); ++y)
  {
  ++localProbability[image.getPixel(x, y)];
  }
}
```

Код слева работает значительно быстрее чем справа (приблизительно в 5 раз). Я это объясняю тем, что внутри image изображение хранится в виде vector<vector<uint8_t>> storage. Для получения пикселя с координатами (x,y) используется storage[y][x]. И во втором случае практически каждый запрос .getPixel(x,y) запрашивал элемент из новой строки => нового вектора, а его могло и не быть в кэше. А в левом случае элементы запрашиваются последовательно, из-за чего кол-во кэш-промахов значительно снижается. Дальше мы ещё вернёмся к этой особенности хранения изображения.

Благодаря этому наблюдению, в будущем будет объяснено отсутвие ожидаемого эффекта от chunk_size.

Параллелизм при фильтрации изображения.

После нахождения лучших порогов их применение заключается также в походе по матрице изображения и присвоении новых значений, в зависимости от изначального значения пикселя, то здесь при распараллеливании работают те же идеи что и при подсчёте диаграммы. Единственное стоит добавить, что для оптимизации фильтрации был создан массив из 256 элементов, где каждый старый оттенок сопоставляется оттенку после фильтрации. И это позволило избавиться от множества условий в дорогом цикле по всему изображению.

Результат работы написанной программы.

CPU: Intel Core i5-10600KF 6 Cores/12 Threads 4.6GHz

Изображение: из репозитория

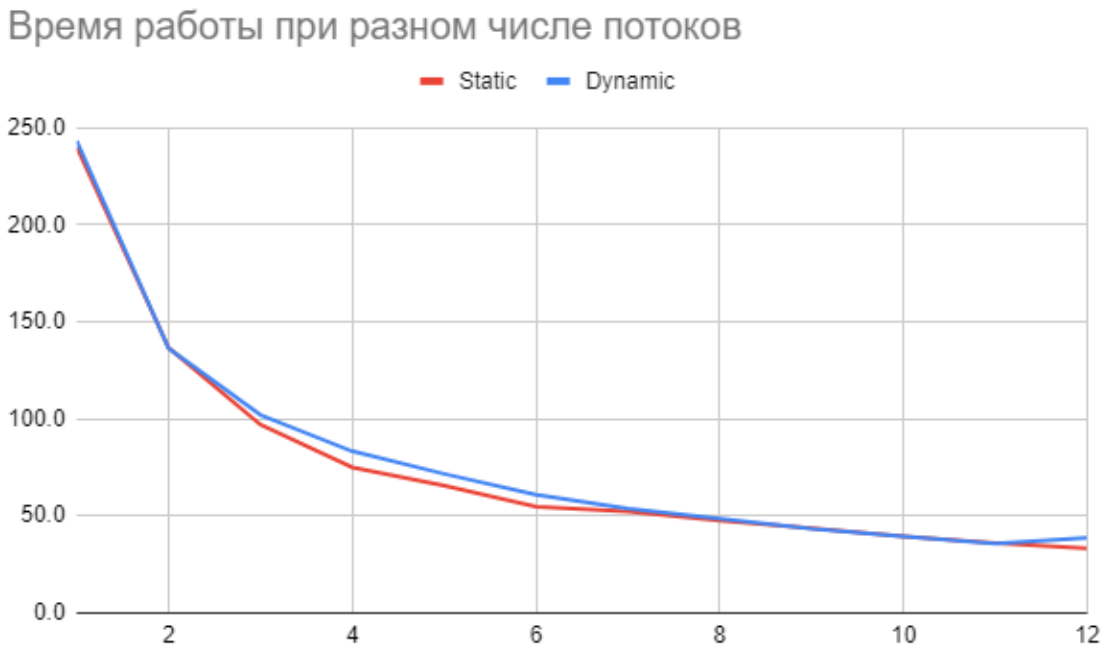
Выбрал на мой взгляд самую удачную версию (schedule static для гистограммы и фильтрации, dynamic для поиска порогов. Везде без chunk_size)

.\omp4.exe -1 .\test_data\in.pgm .\out.pgm	Time (-1 thread(s)): 4.99988 ms 77 130 187
.\omp4.exe 0 .\test_data\in.pgm .\out.pgm	Time (12 thread(s)): 1.99986 ms 77 130 187
.\omp4.exe 1 .\test_data\in.pgm .\out.pgm	Time (1 thread(s)): 4.99988 ms 77 130 187
.\omp4.exe 2 .\test_data\in.pgm .\out.pgm	Time (2 thread(s)): 3.00002 ms 77 130 187

Экспериментальная часть.

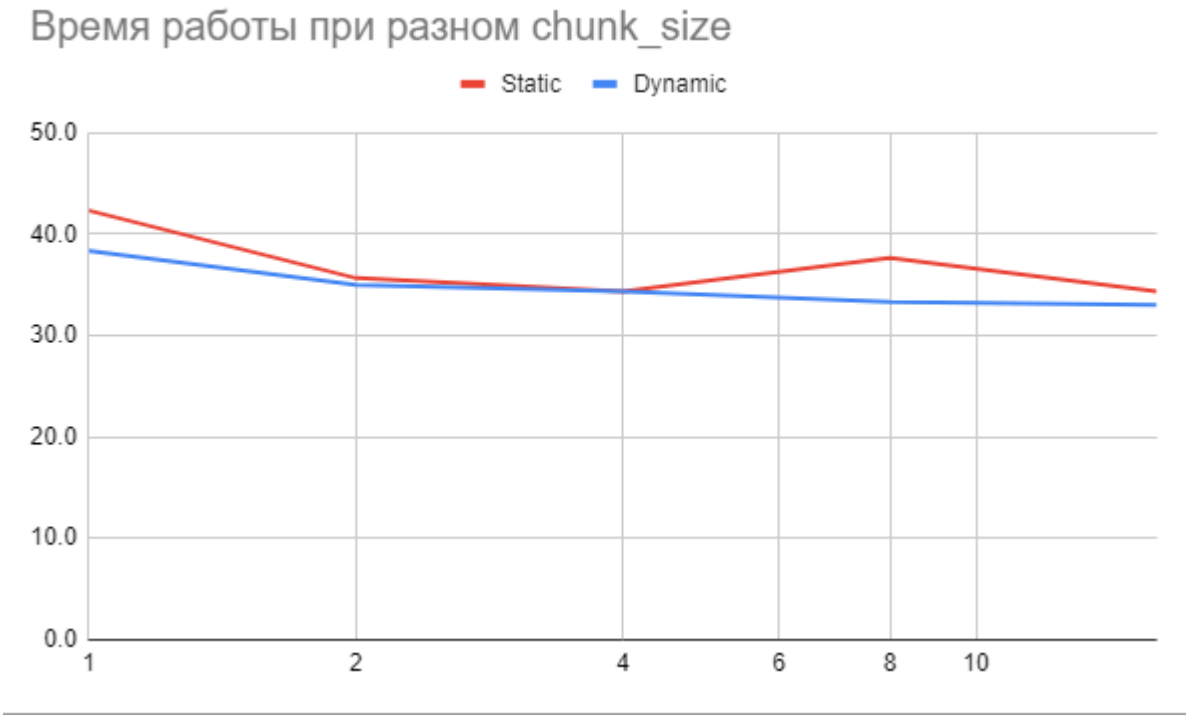
Разное количество потоков.

Программа тестировалась на 16к изображении, потому что на картинке из репозитория программа работает за миллисекунды и относительная погрешность была бы выше. Все замеры проводились по 3 раза и брался средний результат.



На первом графике видно, что от числа потоков программа значительно ускорятся. Что и ожидалось, если работа разделена на 2 ядра то она должна работать примерно в 2 раза быстрее и так далее. Это мы и видим на графике.

Разный chunk_size.



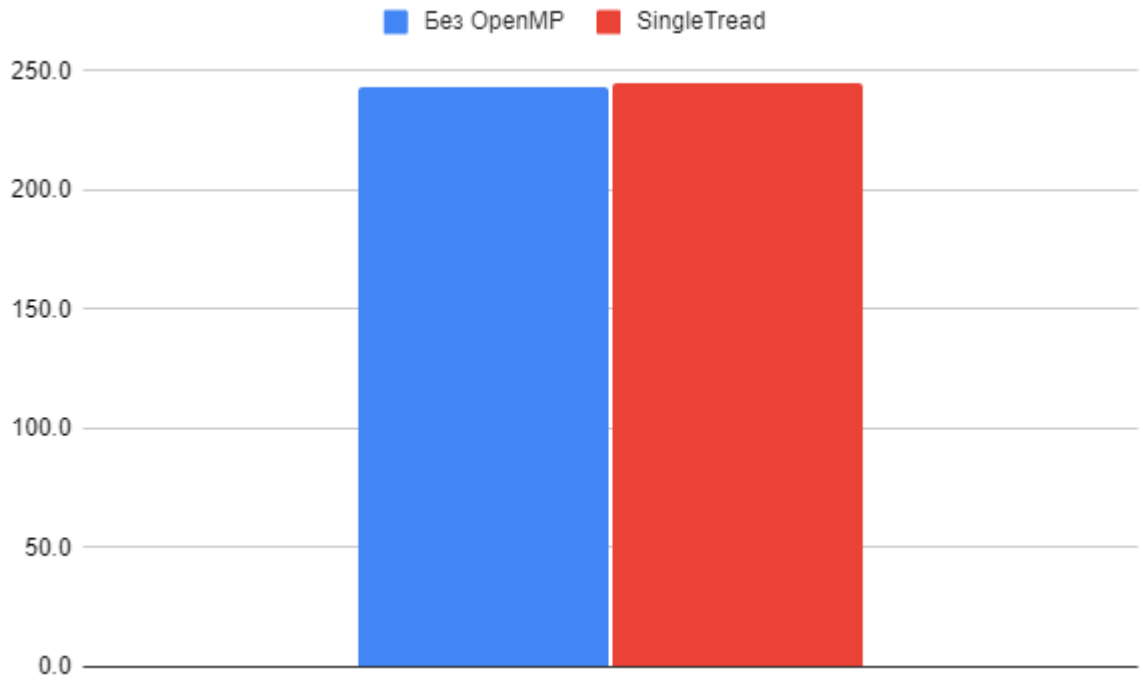
Тесты chunk_size проводились в 12 потоков. Также важно заметить что шкала X логарифмическая.

Ожидалось увидеть сильное замедление при chunk_size = 1 для dynamic, однако его не заметно и для этого есть следующее объяснение.

Если хранит картинку в одномерном массиве длины sizeX*sizeY, и итерироваться по нему одним циклом (который будет параллелиться), то при dynamic с маленьким chunk_size (меньшим чем длинна кэш-линии/sizeof(элемент массива)), то в кэшах нескольких потоков может активно использоваться одна и та же кэш-линия, в разные места которой они будут совершать записи. Записи же будут приводить к инвалидации линии во многих кэшах, что сильно увеличит кол-во кэш-промахов. Данное явление называется False Sharing.

Однако если вспомнить тот способ хранения изображения который используется у меня, то можно заметить одну вещь. Я использую двойной вложенный цикл, распараллеливания верхний (отвечающий за номер строки). При этом разные потоки никогда не будут писать в одну строку изображения. И так как размер строки изображения больше чем размер кэш-линии, то в одну кэш линию они попадать также не будут. Что повышает количество кэш-попаданий и ускоряет программу.

Без OpenMP и Single Thread.



Время работы с 1 потоком и без OpenMP практически не отличается. Для тестов с отключенным OpenMP у pragma omp parallel использовалась директива if, о которой говорилось выше.

Список источников.

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html#:~:text=The%20schedule

https://learn.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170

https://stackoverflow.com/questions/29916732/is-dynamic-scheduling-better-or-static-scheduling-parallel-programming

https://stackoverflow.com/questions/39116329/enable-disable-openmp-locally-at-runtime

https://habr.com/ru/post/187752/

Листинг кода.

hard.cpp

```
#include <iostream>

#include <fstream>

#include <cstring>

#include <vector>



#include "otsuFuncs.h"



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

{

    if (argc < 4)

    {

        std::cout << "3 arguments expected, " + std::to_string(argc - 1) +
" found\n";

        return 0;

    }



    const std::string threadsCountStr = argv[1];

    const std::string in = argv[2];

    const std::string out = argv[3];
```

```
            int threadsCount;
            try
            {
                threadsCount = std::stoi(threadsCountStr);
                if (threadsCount < -1)
                {
                    std::cout << "Threads count " + std::string(threadsCountStr) +
                        " is invalid.\nValid values:\n\t0: Default number of
threads\n      >0: Given number of threads\n       -1: Disable OpenMP
(SingleThread)\n";
                    return 0;
                }
            }
            catch (std::invalid_argument& e)
            {
                std::cout << "Threads count should be a number: " <<
std::string(threadsCountStr) << std::endl;
                return 0;
            }


            try
            {
                PnmImage image;
                image.loadFromFile(in);
```

```
            if (threadsCount > 0)
            {
                omp_set_num_threads(threadsCount);
            }
            const bool isOmpEnabled = threadsCount != -1;

            const double start = omp_get_wtime();
            const std::vector<int> thresholds = calculateOtsuThresholds(image,
isOmpEnabled);
            image.applyThresholds(thresholds, isOmpEnabled);
            const double end = omp_get_wtime();

            printf("Time (%i thread(s)): %g ms\n", threadsCount == 0 ?
omp_get_max_threads() : threadsCount,
                   (end - start) * 1000);
            printf("%u %u %u\n", thresholds[0], thresholds[1], thresholds[2]);

            image.saveToFile(out);
        }
        catch (std::ios_base::failure& e)
        {
            std::cout << e.what() << std::endl;
```

```
        }
        catch (std::runtime_error& e)
        {
            std::cout << e.what() << std::endl;
        }

        return 0;
    }
```

otsuFuncs.cpp

```
    #include "otsuFuncs.h"

    double* calculateProbabilities(const PnmImage& image, const bool
ompEnabled)
    {
        auto* probability = new double[INTENSITY_LAYER_COUNT];
        memset(probability, 0.0, INTENSITY_LAYER_COUNT * sizeof(double));
    #pragma omp parallel if (ompEnabled) default(none) shared(probability,
image)
        {
            auto* localProbability = new double[INTENSITY_LAYER_COUNT];
            memset(localProbability, 0.0, INTENSITY_LAYER_COUNT *
sizeof(double));
```

```
#pragma omp for schedule(static)
        for (int y = 0; y < image.getYSize(); ++y)
        {
            for (int x = 0; x < image.getXSize(); ++x)
            {
                ++localProbability[image.getPixel(x, y)];
            }
        }
#pragma omp critical
        {
            for (int i = 0; i < INTENSITY_LAYER_COUNT; ++i)
            {
                probability[i] += localProbability[i];
            }
        }
        delete[] localProbability;
    }


    const int totalPixelCount = image.getXSize() * image.getYSize();
    for (int i = 0; i < INTENSITY_LAYER_COUNT; ++i)
    {
        probability[i] /= totalPixelCount;
```

```
    }


    return probability;
}


double* calculatePrefOmegas(const double* probability)
{
    auto* omega = new double[INTENSITY_LAYER_COUNT];
    omega[0] = probability[0];
    for (int i = 1; i < INTENSITY_LAYER_COUNT; ++i)
    {
        omega[i] = omega[i - 1] + probability[i];
    }


    return omega;
}


double* calculatePrefMus(const double* probability)
{
    auto* mu = new double[INTENSITY_LAYER_COUNT];
    mu[0] = 0.0;
    for (int i = 1; i < INTENSITY_LAYER_COUNT; ++i)
```

```
            {
                mu[i] = mu[i - 1] + i * probability[i];
            }

            return mu;
        }

        double getPrefOmegaRange(const double* omega, const int left, const int
right)
        {
            return omega[right] - (left >= 0 ? omega[left] : 0.0);
        }

        double getPrefMuRange(const double* mu, const int left, const int right)
        {
            return mu[right] - (left >= 0 ? mu[left] : 0.0);
        }

        double calculateSigmaForClass(const double* prefOmega, const double*
prefMu, const int left, const int right)
        {
            const double omegaRange = getPrefOmegaRange(prefOmega, left, right);
            const double muRange = getPrefMuRange(prefMu, left, right);
```

```
            return muRange * muRange / omegaRange;

        }


        std::vector<int> calculateOtsuThresholds(const PnmImage& image, const bool
ompEnabled)

        {

            const auto* probability = calculateProbabilities(image, ompEnabled);

            const auto* prefOmega = calculatePrefOmegas(probability);

            const auto* prefMu = calculatePrefMus(probability);



            std::vector<std::pair<double, std::vector<int>>> results;

        #pragma omp parallel if (ompEnabled) default(none) shared(prefOmega,
prefMu, results)

            {

                double localBestSigma = 0.0;

                std::vector<int> localBestThresholds(3);

        #pragma omp for schedule(dynamic)

                for (int i = 1; i < INTENSITY_LAYER_COUNT - 3; ++i)

                {

                    const double firstClassSigma =
calculateSigmaForClass(prefOmega, prefMu, -1, i);

                    for (int j = i + 1; j < INTENSITY_LAYER_COUNT - 2; ++j)

                    {
```

```
                        const double secondClassSigma =
calculateSigmaForClass(prefOmega, prefMu, i, j);

                        for (int k = j + 1; k < INTENSITY_LAYER_COUNT - 1; ++k)

                        {

                            const double thirdClassSigma =
calculateSigmaForClass(prefOmega, prefMu, j, k);

                            const double fourthClassSigma = calculateSigmaForClass(

                                prefOmega, prefMu, k, INTENSITY_LAYER_COUNT - 1);



                            const double curSigma = firstClassSigma +
secondClassSigma + thirdClassSigma + fourthClassSigma;

                            if (curSigma > localBestSigma)

                            {

                                localBestSigma = curSigma;



                                localBestThresholds[0] = i;

                                localBestThresholds[1] = j;

                                localBestThresholds[2] = k;

                            }

                        }

                    }

                }



      #pragma omp critical
```

```
        {
            results.emplace_back(localBestSigma, localBestThresholds);
        }
    }

    double overallBestSigma = 0.0;
    int overallBestIndex = 0;
    for (int i = 0; i < results.size(); ++i)
    {
        if (results[i].first > overallBestSigma)
        {
            overallBestSigma = results[i].first;
            overallBestIndex = i;
        }
    }

    delete[] probability;
    delete[] prefOmega;
    delete[] prefMu;

    return results[overallBestIndex].second;
}
```

otsuFuncs.h

```
#pragma once

#include <omp.h>

#include <fstream>

#include <cstring>

#include <vector>

#include "PnmImage.h"



double* calculateProbabilities(const PnmImage& image, bool ompEnabled =
true);



double* calculatePrefOmegas(const double* probability);



double* calculatePrefMus(const double* probability);



double getPrefOmegaRange(const double* omega, int left, int right);



double getPrefMuRange(const double* mu, int left, int right);



std::vector<int> calculateOtsuThresholds(const PnmImage& image, bool
ompEnabled = true);
```

PnmImage.h

```
#pragma once
```

```
#include <fstream>
#include <string>
#include <vector>

constexpr int INTENSITY_LAYER_COUNT = 256;

class PnmImage
{
private:
    int sizeX = 0;
    int sizeY = 0;
    std::vector<std::vector<uint8_t>> storage;

public:
    int getXSize() const
    {
        return sizeX;
    }

    int getYSize() const
    {
        return sizeY;
```

```
}

uint8_t getPixel(const int x, const int y) const
{
    return storage.at(y).at(x);
}

void setPixel(const uint8_t value, const int x, const int y)
{
    storage.at(y).at(x) = value;
}

void loadFromFile(const std::string& path)
{
    std::ifstream input;
    input.open(path, std::ios::binary);
    if (!input.is_open())
    {
        throw std::ios_base::failure("Can`t open input file");
    }

    std::string type;
```

```
                input >> type;
                if (type != "P5")
                {
                    const std::string message = "Unknown File type: '" + type +
"'";
                    throw std::runtime_error(message);
                }

                std::string width;
                input >> width;

                std::string height;
                input >> height;

                int huesCount;
                std::string huesCountStr;
                input >> huesCountStr;
                try
                {
                    sizeX = std::stoi(width);
                    sizeY = std::stoi(height);
                    huesCount = std::stoi(huesCountStr);
                }
```

```
                catch (std::invalid_argument& e)

                {

                    const std::string message = "Width, Height and hues count
should be numbers:\n\tWidth: " + width +

                        "\n\tHeight: " + height +

                        "\n\tHues Count: " + huesCountStr;

                    throw std::runtime_error(message);

                }



                if (sizeX <= 0 || sizeY <= 0)

                {

                    throw std::runtime_error("Width and Height should be
positive:\n\tWidth: " + width +

                        "\n\tHeight: " + height);

                }



                if (huesCount != 255)

                {

                    throw std::runtime_error(

                        "Invalid count of hues: '" + huesCountStr + "'\nExpected:
'" + std::to_string(

                            INTENSITY_LAYER_COUNT - 1) + "'");

                }
```

```
    input.ignore(sizeof(char)); // \n

    storage.clear();
    storage.resize(sizeY);

    for (int y = 0; y < sizeY; ++y)
    {
        for (int x = 0; x < sizeX; ++x)
        {
            uint8_t uintBuff = 0;
            input.read((char*)&uintBuff, sizeof(uint8_t));
            storage.at(y).push_back(uintBuff);
        }
    }
}

void saveToFile(const std::string& path) const
{
    std::ofstream output;
    output.open(path, std::ios::binary);
    if (!output.is_open())
    {
```

```
                throw std::ios_base::failure("Can`t open output file");

            }

            const std::string headerStr = "P5\n" + std::to_string(sizeX) + " "
+ std::to_string(sizeY) + "\n255\n";

            output.write(headerStr.c_str(), headerStr.size());



            for (int y = 0; y < sizeY; ++y)

            {

                for (int x = 0; x < sizeX; ++x)

                {

                    output.write((char*)&storage.at(y).at(x), sizeof(uint8_t));

                }

            }

        }



        void applyThresholds(const std::vector<int>& thresholds, const bool
ompEnabled = true)

        {

            auto* classes = new uint8_t[INTENSITY_LAYER_COUNT];

            fillClassesArray(classes, thresholds);

    #pragma omp parallel if (ompEnabled)

            {

    #pragma omp for schedule(static)
```

```
                for (int y = 0; y < getYSize(); ++y)
                {
                    for (int x = 0; x < getXSize(); ++x)
                    {
                        setPixel(classes[getPixel(x, y)], x, y);
                    }
                }
            }

        private:
            static void fillClassesArray(uint8_t* classes, const std::vector<int>&
thresholds)
            {
                for (int i = 0; i <= thresholds[0]; ++i)
                {
                    classes[i] = 0;
                }

                for (int i = thresholds[0] + 1; i <= thresholds[1]; ++i)
                {
                    classes[i] = 84;
                }
```

```
                for (int i = thresholds[1] + 1; i <= thresholds[2]; ++i)
                {
                    classes[i] = 170;
                }

                for (int i = thresholds[2] + 1; i <= INTENSITY_LAYER_COUNT - 1;
++i)
                {
                    classes[i] = 255;
                }
            }
        };
```