САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

«OpenMP»

Выполнил: Трофимов Никита Сергеевич

Номер ИСУ: 334969

студ. гр. М3139

Санкт-Петербург

2021

Цель работы: знакомство со стандартом OpenMP.

Инструментарий и требования к работе: стандарт OpenMP 2.0, C++, стандарт *gnu++17*.

I. Теоретическая часть.

OpenMP – библиотека для параллельного программирования вычислительных систем с общей памятью. По началу выполняется только один процесс, как обычная программа, но встретив параллельную область (`#pragma omp parallel`), порождается несколько потоков (их можно задать вручную, по умолчанию – кол-во вычислительных ядер в системе). На *Рисунок №1* и *Рисунок №2* я привёл схематичный принцип работы OpenMP.

Рисунок №1.

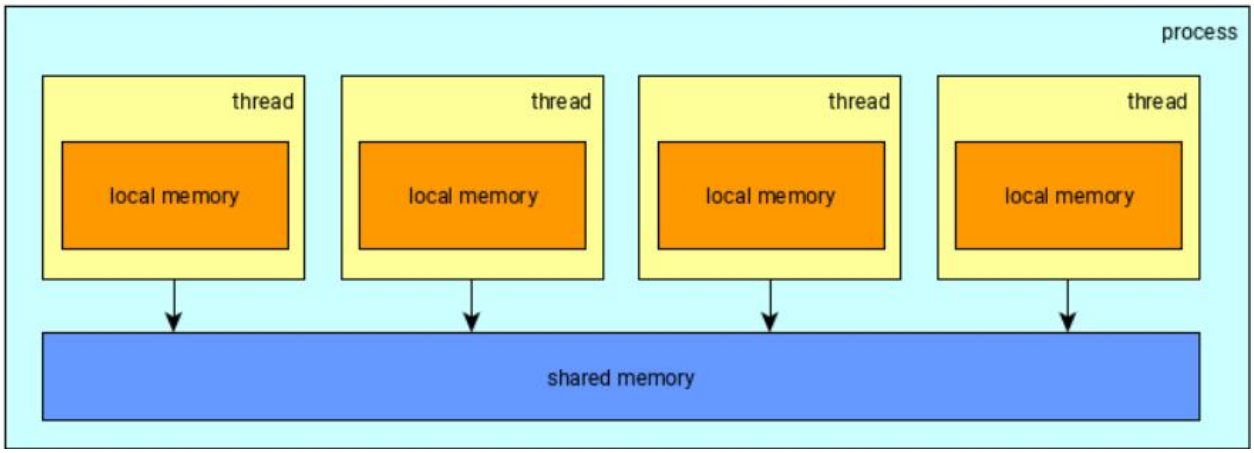
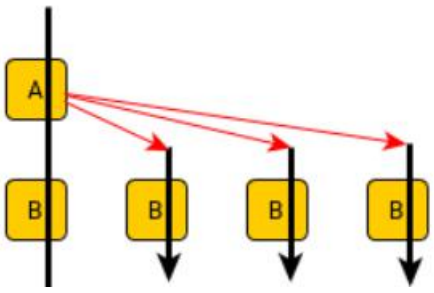


Рисунок №2.



Существует проблема, что несколько потоков будут в каком-то рандомном порядке обращаться к их общей переменной (запись, чтение, изменение, etc.), а следовательно результат может быть

неожиданным. Для этого есть две директивы: `#pragma omp atomic` и `#pragma omp critical`.

Если цикл можно разделить между потоками, то есть его итерации независимы между друг другом, то такой цикл называется векторизуемым.

Опции планирования для цикла `schedule:`

- `static` – статическое планирование. Итерации цикла будут поделены приблизительно поровну между потоками. Вторым параметром можно конкретно задать сколько итераций мы хотим.
- `dynamic` – динамическое планирование. По умолчанию кол-во итераций равно 1, но его тоже можно задать вручную. В отличие от статического планирования здесь распределение между потоками зависит от трудоёмкости каждой конкретной итерации.

Есть такая опция `reduction(+: variable)` – она проводит редукцию в цикле при вычислении, например, суммы. То есть в каждом потоке создаётся локальная переменная, в которую и считается частная сумма, потом в ответ все эти частные переменные просто суммируются. Это можно делать с *, -, etc.

Для сокращения кода, его простоты и лучшего понимания всё можно писать в одну строчку: `#pragma omp parallel for`. Существует опция `shared`, в параметрах к которой мы передаем переменные из внешнего окружения, которые мы хотим передать внутрь.

Есть опция `num_threads`, которая вручную задаёт кол-во потоков.

Типы файлов для хранения изображений: `*.pgm` или `*.ppm`. Про первый: в нём хранится черно-белое изображение (разные оттенки серого от 0 до 255), он начинается с P5, для каждого пикселе выделяется ровно один байт. Про второй формат: в нём хранится цветное изображение в формате RGB – три последовательных байта для трёх соответственно компонент. Начинается файл с P6. За счёт прямого хранения пикселей мы получаем простую работу с файлом, но при этом он становится

очень большим, что влияет на время работы программы, для этого мы и будем использовать OpenMP.

II. Практическая часть.

Для корректной работы программы нужна библиотека <omp.h>. Чтобы скомпилировать программу нужно добавить опцию компилятора -fopenmp. Чтобы добавить её в Clion, я добавил в CMakeLists такую строчку: `set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp")`.

Про реализацию OpenMP в моём коде:

* У меня в программе есть три области, которые занимают много времени и которые требуют распараллеливания. (за исключением ввода и вывода)
* Первая. Я считаю массив `cnt[i]` – количество пикселей с цветом `i`. Для этого я задаю кол-во потоков функцией `omp_set_num_threads`, затем я создаю параллельную область, в которой я выполняю подсчёт. Как работает параллельная область? Я создаю локальный массив `cnt` на 256 элементов в каждом потоке и этот поток в нём считает. Далее я в области `critical` сливаю этот локальный массив с основным. Это всего 256 итераций, что совсем немного. При этом локальные массивы заполняются параллельно. В итоге:

```
#pragma omp parallel default(none) shared(data, block, countColors)
#pragma omp critical
```

* Вторая. Эта и следующая параллельные области намного проще в реализации. Для подсчёта переменной `center` (это просто подсчёт суммы, про неё написано далее) я использовал редукцию. Реализация в одну строчку, а идея такая: каждый поток считает сумму в локальной переменной, а потом складывает всё в одну. В итоге:

```
#pragma omp parallel for default(none) shared(cols, rows, data) num_threads(numberOfThreads) reduction(+:center) schedule(static)
```

* Третья. Теперь нужно записать новые значения цветов опять в массив с картинкой. Здесь итерации цикла не зависят друг от друга, поэтому простое распараллеливание цикла уже будет работать. (без `critical` и `reduction`).

  В итоге:

  ```
  #pragma omp parallel for default(none) shared(cols, rows, data, convertedColorsR, convertedColorsG, convertedColorsB) schedule(static)
  ```

* Это описано для одного канала. Для RGB аналогично.

Для ввода/вывода я использовал FILE *file и две функции: `fgetc` и `fputc` – для чтения и записи соответственно. Чтение я осуществлял побайтово, для типа `Byte` я создал `typedef` для `unsigned char`. В моём коде есть обработка всяческих типов ошибок: неправильное расширение файлов, неправильный коэффициент или кол-во потоков. Также при чтении файла, я проверяю, что он начинается на P5 или P6, что максимальное значение для цвета – 255.

Для точного замера времени я использую библиотеку `<chrono>`.

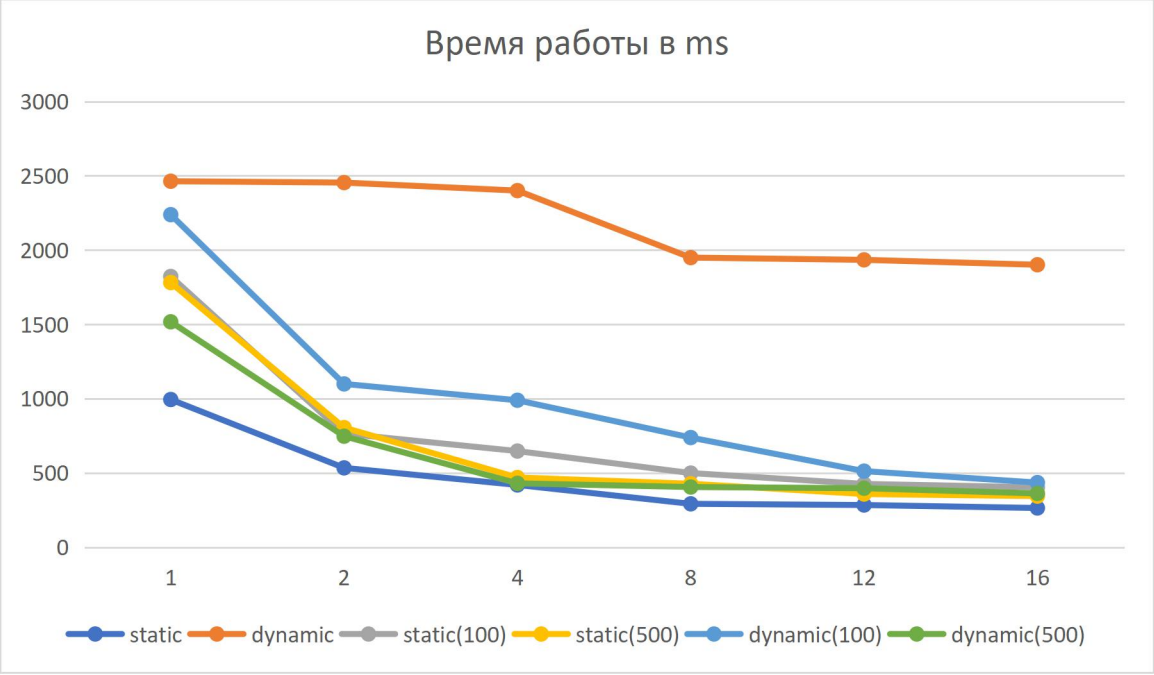
Теперь про то, как я реализовал увеличение контрастности:

- Я считаю массив `countColors`, в котором считаю встречаемость каждого из цветов от 0 до 255, ещё я храню переменную `center`, в которой считаю среднее значение цвета пикселя (считаю сумму, потом делю на `cols * rows`).
- Дальше я отсчитываю два цвета: до которого и с которого игнорировать цвета – коэффициент `k`. Получаю два числа: `ignore_less` и `ignore_more`.

- `center` – это “центр” всех пикселей, среднее значение. В идеально сбалансированной картинке это значение равно 127, но в реальности это не так, поэтому я его и считаю.
- Как я растягиваю значения? Во-первых, я растягиваю их с насыщением, то есть, если i < `ignore_less` или i > `ignore_more`, то я ставлю значения этих пикселей в 0 и 255 соответственно. В противном случае я применяю две формулы (для пикселей левее среднего значения и правее): $center * \frac{i - \text{ignore\_less}}{center - ignore\_less}$ и $\frac{255 * (\text{i} - \text{center}) - \text{center} * \text{i} + \text{center} * \text{ignore\_more}}{\text{ignore\_more} - \text{center}}$. Эти формулы я получил из подобия треугольников. Отрезок `[x, y]` растягиваем в `[0, 255]`, берём точку `z` и получаем из неё `z’`. Не забываем, что значение в средней точке не меняется.
- Данный алгоритм нужно применить для серой картинки именно в таком виде. Если картинка цветная, то применим его для каждого из каналов R, G, B по отдельности.

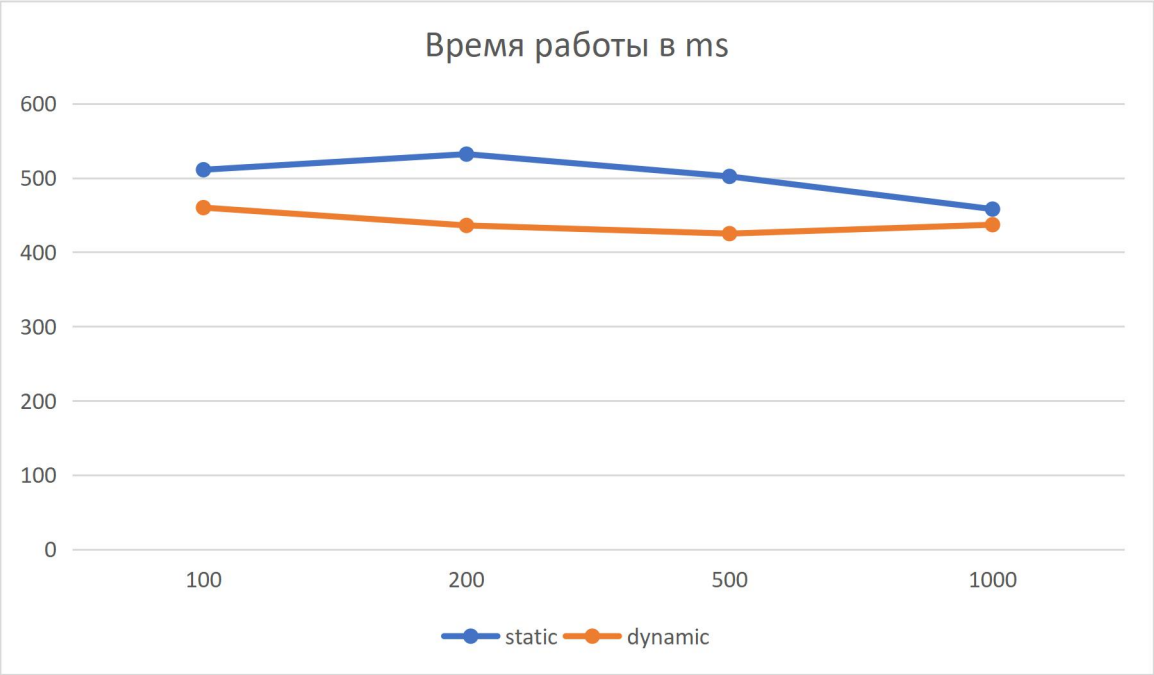
Графики:

❖ `threads` = {1, 2, 4, 8, 12, 16}, `schedule` = {‘`static`’, ‘`dynamic`’}, `chunk_size` = {`default`, 100, 500}



Вывод: если просто выбирать между static и dynamic, то явно лучше выбрать static, он работает намного быстрее. В противном случае они идут близко друг к другу, периодически обгоняя один другого.

- `threads` = {4}, `schedule` = {‘`static`’, ‘`dynamic`’}, `chunk_size` = {‘100’, ‘200’, ‘500’, ‘1000’}



Вывод: если можно варьировать параметры у `static` и `dynamic`, то здесь выигрывает dynamic (на значении 500), конечно не сильно, но всё же.

- OpenMP с одним потоком и просто без OpenMP

  В один поток: 1650 ms.

  Без OpenMP: 1280 ms.

  Вывод: с одним потоком явно лучше без OpenMP. Программа и так работает в один поток, а с OpenMP ей ещё нужно создать какие-то дополнительные вещи, что тормозит работу программы.

III. Листинг кода:

<main.cpp>

```
#include <iostream>
#include <cstdio>
#include <vector>
#include <cstdlib>
#include <chrono>
#include <algorithm>
#include <omp.h>

using namespace std;

typedef unsigned char Byte;

const Byte space = ' ';
const Byte lineSeparator = '\n';
int numberOfThreads;
double contrastRatio;

enum FileFormats {
    PGM, PPM, WRONG
};

int cols, rows;

struct RGB {
    Byte R{}, G{}, B{};

    RGB() = default;

    RGB(Byte r, Byte g, Byte b) : R(r), G(g), B(b) {}
};

void convertAllRGBBytes(FILE *input, FILE *output) {
    cout << "Reading image..." << "\n";
    cout.flush();
    vector<RGB> data(cols * rows);
    for (int i = 0; i < cols * rows; i++) {
        Byte r = fgetc(input), g = fgetc(input), b = fgetc(input);
        data[i] = RGB(r, g, b);
    }
    cout << "Image was read." << "\n";

    auto begin = chrono::steady_clock::now();
    cout << "Converting... Wait please!" << "\n";
    cout.flush();

    vector<Byte> convertedColorsR = vector<Byte>(256);
    vector<Byte> convertedColorsG = vector<Byte>(256);
    vector<Byte> convertedColorsB = vector<Byte>(256);
    vector<int> countColorsR = vector<int>(256);
    vector<int> countColorsG = vector<int>(256);
    vector<int> countColorsB = vector<int>(256);
    long long centerR = 0, centerG = 0, centerB = 0;

    omp_set_num_threads(numberOfThreads);
    long long block = cols * rows / numberOfThreads;
#pragma omp parallel default(none) shared(data, block, countColorsR, countColorsG,
countColorsB)
    {
        vector<int> tmpCountColorsR = vector<int>(256);
```

```
        vector<int> tmpCountColorsG = vector<int>(256);
        vector<int> tmpCountColorsB = vector<int>(256);
        int cur = omp_get_thread_num();
        long long from = cur * block, to = (cur + 1) * block - 1;
        for (long long i = from; i <= to; i++) {
            tmpCountColorsR[data[i].R]++;
            tmpCountColorsG[data[i].G]++;
            tmpCountColorsB[data[i].B]++;
        }

        for (int i = 0; i < 256; i++) {
#pragma omp critical
            countColorsR[i] += tmpCountColorsR[i];
#pragma omp critical
            countColorsG[i] += tmpCountColorsG[i];
#pragma omp critical
            countColorsB[i] += tmpCountColorsB[i];
        }
    }

#pragma omp parallel for default(none) shared(cols, rows, data)
num_threads(numberOfThreads) reduction(+:centerR, centerG, centerB) schedule(static)
    for (int i = 0; i < cols * rows; i++) {
        centerR += data[i].R;
        centerG += data[i].G;
        centerB += data[i].B;
    }

    centerR /= cols * rows;
    centerG /= cols * rows;
    centerB /= cols * rows;
    int ignore_lessR = 0, ignore_moreR = 255;
    int ignore_lessG = 0, ignore_moreG = 255;
    int ignore_lessB = 0, ignore_moreB = 255;
    long long totalR = 0, totalG = 0, totalB = 0, size = cols * rows;
    for (int i = 0; i < 256; i++) {
        totalR += countColorsR[i];
        if ((double) (totalR) / (double) (size) >= contrastRatio) {
            ignore_lessR = i;
            break;
        }
    }
    for (int i = 0; i < 256; i++) {
        totalG += countColorsG[i];
        if ((double) (totalG) / (double) (size) >= contrastRatio) {
            ignore_lessG = i;
            break;
        }
    }
    for (int i = 0; i < 256; i++) {
        totalB += countColorsB[i];
        if ((double) (totalB) / (double) (size) >= contrastRatio) {
            ignore_lessB = i;
            break;
        }
    }
    totalR = 0;
    for (int i = 255; i >= 0; i--) {
        totalR += countColorsR[i];
        if ((double) (totalR) / (double) (size) >= contrastRatio) {
            ignore_moreR = i;
            break;
```

```
        }
    }
    totalG = 0;
    for (int i = 255; i >= 0; i--) {
        totalG += countColorsG[i];
        if ((double) (totalG) / (double) (size) >= contrastRatio) {
            ignore_moreG = i;
            break;
        }
    }
    totalB = 0;
    for (int i = 255; i >= 0; i--) {
        totalB += countColorsB[i];
        if ((double) (totalB) / (double) (size) >= contrastRatio) {
            ignore_moreB = i;
            break;
        }
    }
    for (int i = 0; i < centerR; i++) {
        if (i < ignore_lessR) {
            convertedColorsR[i] = 0;
        } else {
            convertedColorsR[i] = centerR * (i - ignore_lessR) / (centerR -
ignore_lessR);
        }
    }
    for (int i = 0; i < centerG; i++) {
        if (i < ignore_lessG) {
            convertedColorsG[i] = 0;
        } else {
            convertedColorsG[i] = centerG * (i - ignore_lessG) / (centerG -
ignore_lessG);
        }
    }
    for (int i = 0; i < centerB; i++) {
        if (i < ignore_lessB) {
            convertedColorsB[i] = 0;
        } else {
            convertedColorsB[i] = centerB * (i - ignore_lessB) / (centerB -
ignore_lessB);
        }
    }
    for (long long i = centerR + 1; i < 255; i++) {
        if (i > ignore_moreR) {
            convertedColorsR[i] = 255;
        } else {
            convertedColorsR[i] = (255 * (i - centerR) - centerR * i + centerR *
ignore_moreR) /
                                  (ignore_moreR - centerR);
        }
    }
    for (long long i = centerG + 1; i < 255; i++) {
        if (i > ignore_moreG) {
            convertedColorsG[i] = 255;
        } else {
            convertedColorsG[i] = (255 * (i - centerG) - centerG * i + centerG *
ignore_moreG) /
                                  (ignore_moreG - centerG);
        }
    }
    for (long long i = centerB + 1; i < 255; i++) {
        if (i > ignore_moreB) {
```

```
            convertedColorsB[i] = 255;
        } else {
            convertedColorsB[i] = (255 * (i - centerB) - centerB * i + centerB *
ignore_moreB) /
                                  (ignore_moreB - centerB);
        }
    }

    convertedColorsR[centerR] = centerR;
    convertedColorsG[centerG] = centerG;
    convertedColorsB[centerB] = centerB;

#pragma omp parallel for default(none) num_threads(numberOfThreads) shared(cols, rows,
data, convertedColorsR, convertedColorsG, convertedColorsB) schedule(static)
    for (int i = 0; i < cols * rows; i++) {
        data[i].R = convertedColorsR[data[i].R];
        data[i].G = convertedColorsG[data[i].G];
        data[i].B = convertedColorsB[data[i].B];
    }

    auto end = chrono::steady_clock::now();
    auto working_time = chrono::duration_cast<chrono::milliseconds>(end - begin);
    printf("Time (%i thread(s)): %g ms\n", numberOfThreads, working_time.count());

    cout << "Writing image..." << "\n";
    for (int i = 0; i < cols * rows; i++) {
        fputc(data[i].R, output);
        fputc(data[i].G, output);
        fputc(data[i].B, output);
    }
    cout << "Image was writen." << "\n";
}

void convertAllGrayBytes(FILE *input, FILE *output) {
    cout << "Reading image..." << "\n";
    cout.flush();
    vector<Byte> data(cols * rows);
    for (int i = 0; i < cols * rows; i++) {
        Byte c = fgetc(input);
        data[i] = c;
    }
    cout << "Image was read." << "\n";

    auto begin = chrono::steady_clock::now();
    cout << "Converting... Wait please!" << "\n";
    cout.flush();

    vector<Byte> convertedColors = vector<Byte>(256);
    vector<int> countColors = vector<int>(256);
    long long center = 0;

    omp_set_num_threads(numberOfThreads);
    long long block = cols * rows / numberOfThreads;
#pragma omp parallel default(none) shared(data, block, countColors)
    {
        vector<int> tmpCountColors = vector<int>(256);
        int cur = omp_get_thread_num();
        long long from = cur * block, to = (cur + 1) * block - 1;
        for (long long i = from; i <= to; i++) {
            tmpCountColors[data[i]]++;
        }
```

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

#pragma omp parallel for default(none) shared(cols, rows, data)
num_threads(numberOfThreads) reduction(+:center) schedule(static)
    for (int i = 0; i < cols * rows; i++) {
        center += data[i];
    }

    center /= cols * rows;
    int ignore_less = 0, ignore_more = 255;
    long long total = 0, size = cols * rows;
    for (int i = 0; i < 256; i++) {
        total += countColors[i];
        if ((double) (total) / (double) (size) >= contrastRatio) {
            ignore_less = i;
            break;
        }
    }
    total = 0;
    for (int i = 255; i >= 0; i--) {
        total += countColors[i];
        if ((double) (total) / (double) (size) >= contrastRatio) {
            ignore_more = i;
            break;
        }
    }
    for (int i = 0; i < center; i++) {
        if (i < ignore_less) {
            convertedColors[i] = 0;
        } else {
            convertedColors[i] = center * (i - ignore_less) / (center - ignore_less);
        }
    }
    for (long long i = center + 1; i < 255; i++) {
        if (i > ignore_more) {
            convertedColors[i] = 255;
        } else {
            convertedColors[i] = (255 * (i - center) - center * i + center *
ignore_more) /
                                 (ignore_more - center);
        }
    }

    convertedColors[center] = center;

#pragma omp parallel for default(none) num_threads(numberOfThreads) shared(cols, rows,
data, convertedColors) schedule(static)
    for (int i = 0; i < cols * rows; i++) {
        data[i] = convertedColors[data[i]];
    }

    auto end = chrono::steady_clock::now();
    auto working_time = chrono::duration_cast<chrono::milliseconds>(end - begin);
    printf("Time (%i thread(s)): %g ms\n", numberOfThreads, working_time.count());

    cout << "Writing image..." << "\n";
    for (int i = 0; i < cols * rows; i++) {
        fputc(data[i], output);
```

```
    }
    cout << "Image was writen." << "\n";
}

int getNumber(FILE *file) {
    Byte currentByte = fgetc(file);
    string answer;
    while (currentByte != space && currentByte != lineSeparator) {
        answer += (char) currentByte;
        currentByte = fgetc(file);
    }
    char *hole;
    return strtol(answer.c_str(), &hole, 10);
}

FileFormats checkFileFormat(const string &fileName) {
    int sizeOfFile = (int) fileName.length();
    string fileFormat = fileName.substr(sizeOfFile - 4, 4);
    if (sizeOfFile < 5) return WRONG;
    if (fileFormat == ".pgm") return PGM;
    if (fileFormat == ".ppm") return PPM;
    return WRONG;
}

bool checkDigit(const string &number) {
    return all_of(number.begin(), number.end(), [](const char &it) {
        return '0' <= it && it <= '9';
    });
}

bool checkDoubleDigit(const string &number) {
    if (number.length() == 1 && number == "0") {
        return true;
    }
    return number[0] == '0' && number[1] == '.' && checkDigit(number.substr(2,
number.size() - 2));
}

int main(int argc, char *argv[]) {
    if (argc != 5) {
        cout << "Expected 5 argc but found " << argc << "\n";
        return 0;
    }
    char *hole;
    if (!checkDigit(string(argv[1]))) {
        cout << "Expected number of threads - integer, not " << argv[1] << "\n";
        return 0;
    }
    numberOfThreads = strtol(argv[1], &hole, 10);
    cout << "Number of threads: " << numberOfThreads << "\n";
    if (numberOfThreads == 0) {
        numberOfThreads = 16;
    }
    cout << "Input file: " << argv[2] << "\n";
    cout << "Output file: " << argv[3] << "\n";
    if (!checkDoubleDigit(argv[4])) {
        cout << "Contrast ratio expected in [0.0, 0.5) - double " <<
             "but was given " << argv[4] << "\n";
        return 0;
    }
    contrastRatio = strtod(argv[4], &hole);
    if (!(contrastRatio >= 0 && contrastRatio < 0.5)) {
```

```
    cout << "Contrast ratio expected in [0.0, 0.5) " <<
         "but was given " << contrastRatio << "\n";
    return 0;
}
cout << "Contrast ratio: " << contrastRatio << "\n";

FILE *input = fopen(argv[2], "rb");
if (!input) {
    cout << "File " << argv[2] << " doesn't exists!" << "\n";
    return 0;
}

FileFormats first = checkFileFormat(string(argv[2]));
FileFormats second = checkFileFormat(string(argv[3]));
if (first == WRONG || second == WRONG || first != second) {
    cout << "Wrong file format! Or formats mismatch!" << "\n";
    return 0;
}
FileFormats format = first;

vector<Byte> data = vector<Byte>(3);
data[0] = fgetc(input);
data[1] = fgetc(input);
data[2] = fgetc(input);
cout << "File format: " << data[0] << data[1] << "\n";
bool headerData = data[0] == 'P' &&
                  (data[1] == '6' || data[1] == '5');
if (!headerData) {
    cout << "Expected " <<
         (format == PPM ? "P6" : "P5") << ".\n";
    return 0;
}
cols = getNumber(input);
rows = getNumber(input);
cout << "Dimensions: " << cols << " " << rows << "\n";
int maxColor = getNumber(input);
cout << "Max color value: " << maxColor << "\n";
if (maxColor != 255) {
    cout << "Max color value expected 255 but found " <<
         maxColor << "\n";
    return 0;
}

FILE *output = fopen(argv[3], "wb");
if (!output) {
    cout << "File " << argv[3] << " doesn't exists!" << "\n";
    return 0;
}

string info = (format == PPM ? "P6\n" : "P5\n")
              + to_string(cols) + " " + to_string(rows) +
              "\n" + to_string(maxColor) + "\n";
fwrite(info.c_str(), sizeof(char), info.size(), output);

if (format == PPM) {
    convertAllRGBBytes(input, output);
} else {
    convertAllGrayBytes(input, output);
}

fclose(input);
fclose(output);
```

```
    cout << "All is done!" << "\n";

    return 0;
}
```