САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

# «OpenMP»

Выполнил(а): Рынк Артур Эдуардович

Номер ИСУ: 334839

студ. гр. M3135

Санкт-Петербург

2022

**Цель работы:** знакомство со стандартом OpenMP.

**Инструментарий и требования к работе:** рекомендуется использовать C, C++. Возможно использовать Python и Java. Стандарт OpenMP 2.0.

## Теоретическая часть

OpenMP - набор инструментов, позволяющий программисту легко (так как от него не требуется сильно изменять код, дополнять его) распараллеливать свой код (параллельно исполнять разные фрагменты кода) на языках C, C++, Fortran. В стандарт OpenMP входят спецификации набора директив компилятора (основанные на #pragma директивах), процедур (например, количество создаваемых потоков, которое можно задать через библиотечные процедуры) и переменных среды.

Мы будем использовать следующие директивы, процедуры и переменные среды:

1) num_threads (задается с помощью omp_set_num_threads) - количество потоков;
2) for - распараллеливание цикла for (#pragma omp parallel for …);
3) sections - позволяет разбивать код на секции, выполняющиеся параллельно (#pragma omp parallel sections … { #pragma section{...} …});
4) private - для объявления локальных переменных;
5) shared - глобальные переменные (общие для всех потоков);
6) schedule - определяет, как разделить потоки между ядрами, имеет 2 параметра: тип разделения и chunk_size (или default):
    a) static - делит блоки потоков (размером chunk_size) по очереди.
    b) dynamic - делит блоки потоков (размером chunk_size) в каком-то порядке.

7) default - задает какой-то модификатор всем переменным (мы используем default(none), так как сами задаём модификации)

## Практическая часть

1. Описание работы:
   1.1. Сначала мы прочитали файл, нашли там тип файла (PPM или PGM), максимальное значение пикселей (maxVal), размер (width * height). Установили количество потоков (countThreads), узнали имена входного (inFileName) и выходного файла (outFileName), также добавили обработку ошибок.
   1.2. По типу файла (PGM или PPM) мы считываем данные в data (для PPM в dataR, dataG, dataB).
   1.3. Делаем для них сортировку подсчетом (используя sortR), для каждого цвета находим границы значений, которые мы игнорируем (downVal*, upVal*). Далее, используя формулу:

$$newColor = \frac{(oldColor - downColor) * maxVal}{upVal - downVal}$$

   находим новое значение пикселей и записываем их обратно в data (если upVal == downVal (то есть одноцветная картинка), то оставляем ее как есть, также учитываем, чтобы проигнорированные значения не вылетели из границ (у нас арифметика с насыщением)).
   1.4. Записываем тип файла, размер (width и height), maxVal и data в новый файл, сначала все удалив оттуда.
2. Графики:

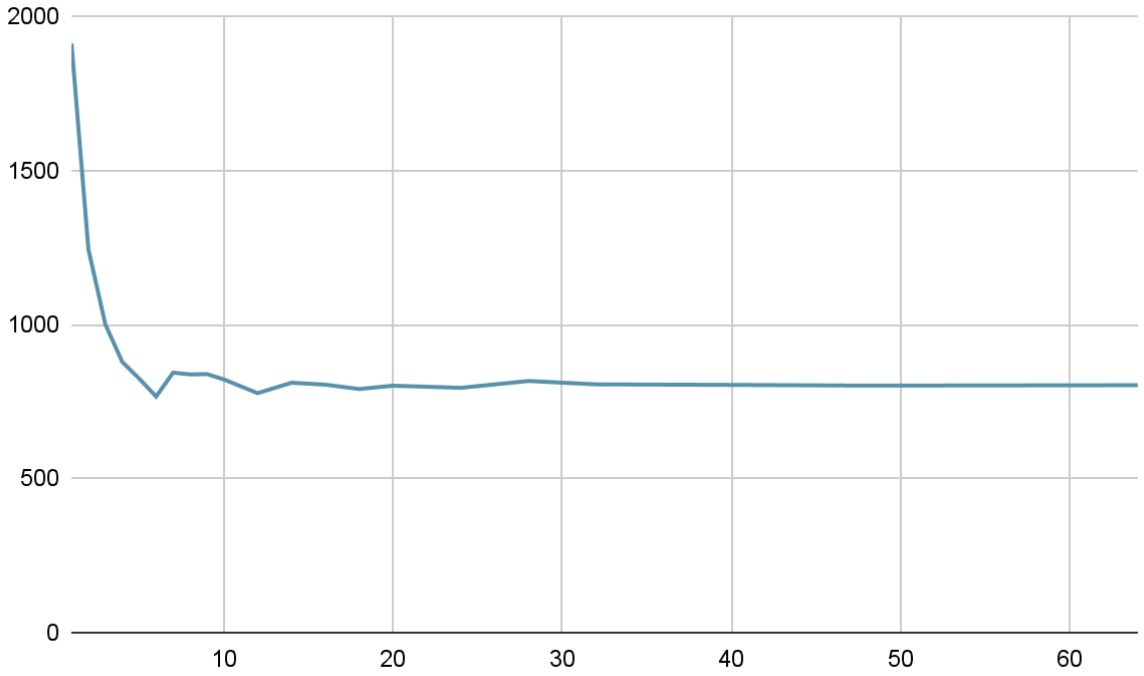


Рисунок №1 – при различных значениях числа потоков при одинаковом параметре schedule* - static и дефолтным chunk_size.

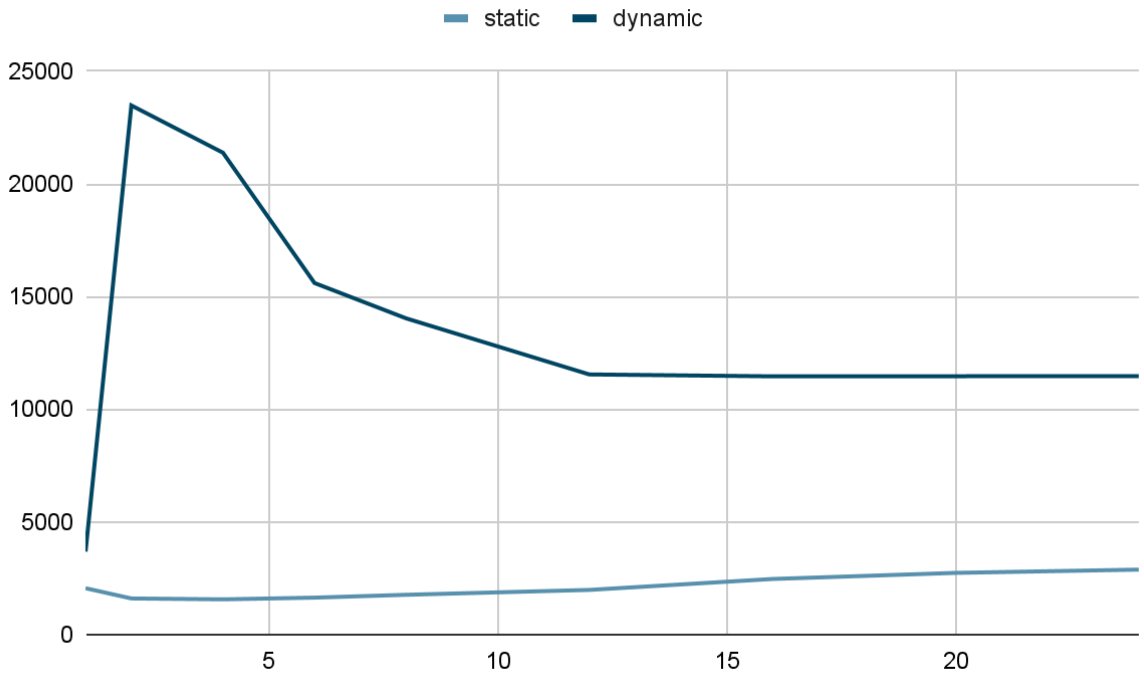


Рисунок №2 – при различных значениях числа потоков при одинаковом параметре schedule* - static и dynamic при chunk_size = 1.

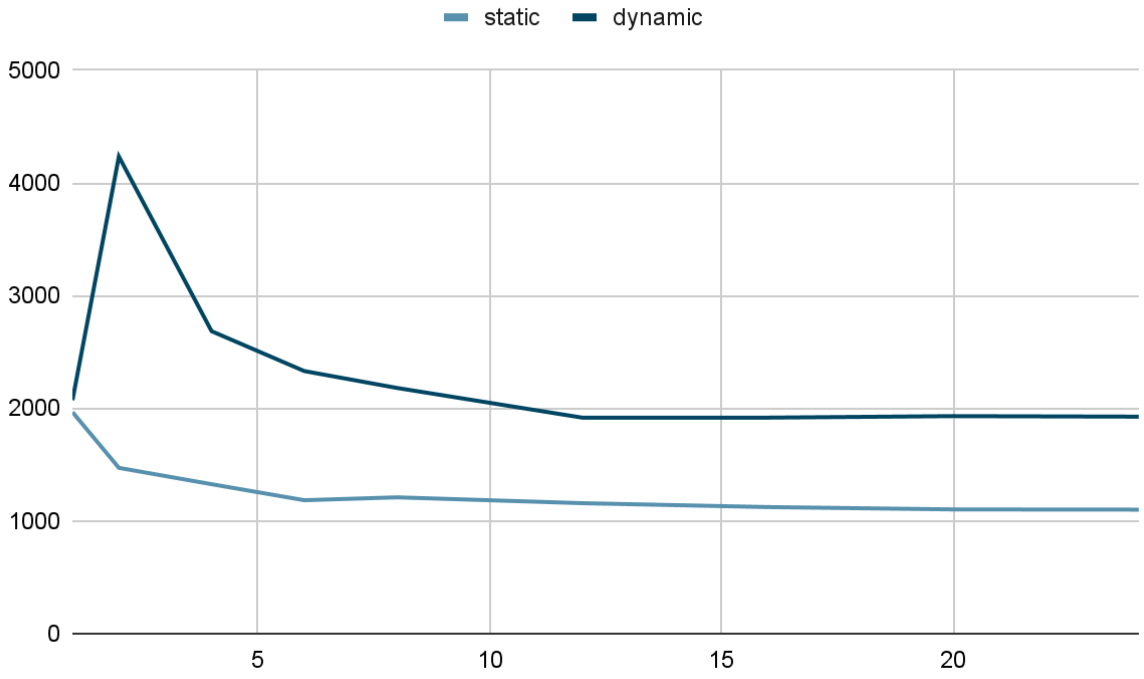


Рисунок №3 – при различных значениях числа потоков при одинаковом параметре schedule* - static и dynamic при chunk_size = 16.

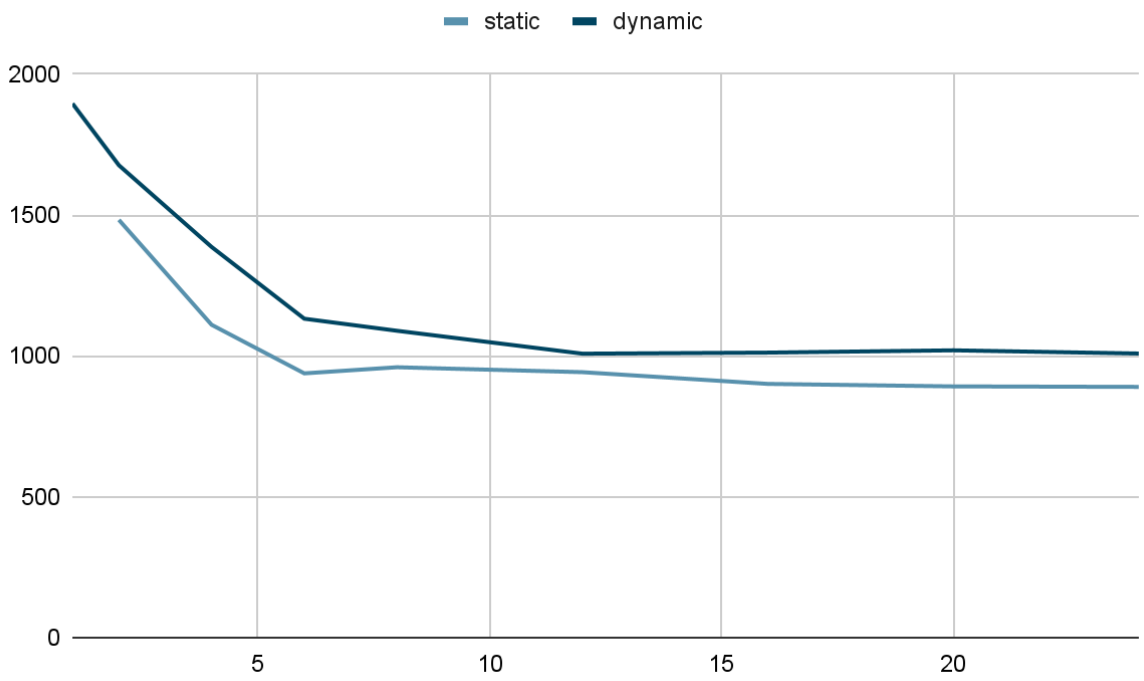


Рисунок №4 – при различных значениях числа потоков при одинаковом параметре schedule* - static и dynamic при chunk_size = 256.

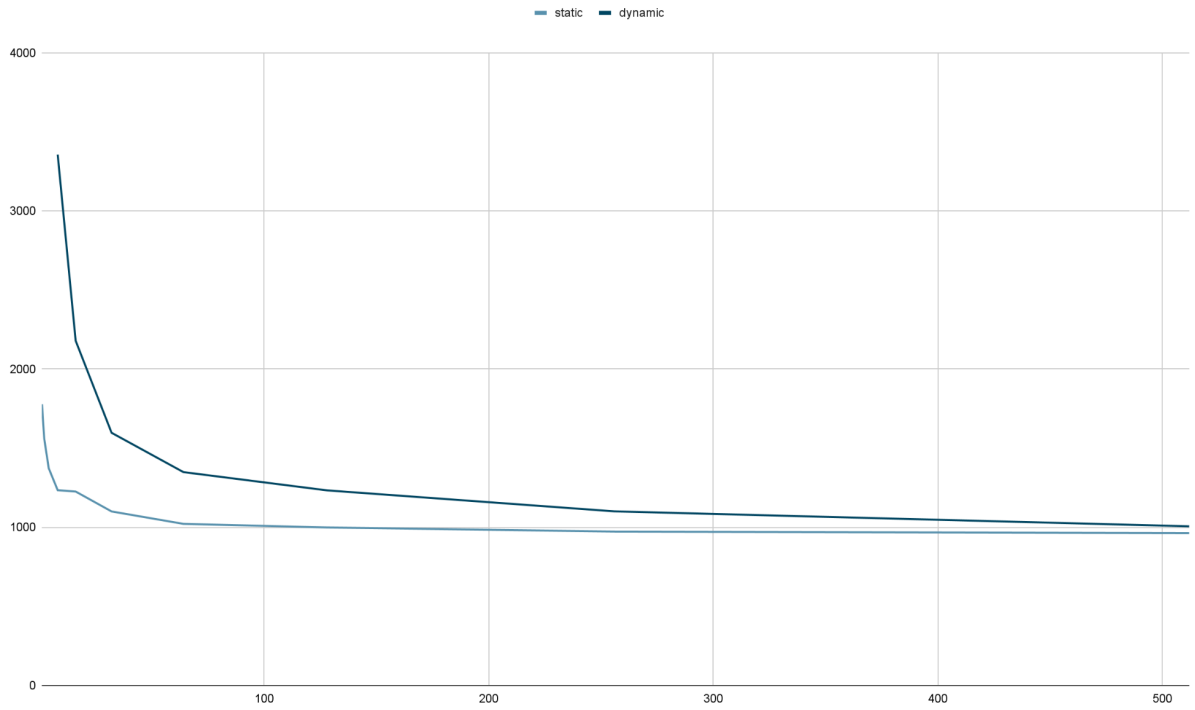


Рисунок №4 – при одинаковом значении числа потоков (8) при различных параметрах schedule* (static и dynamic, chunk_size in [1; 512]);

Время с включенным 1 потоком - 1933.84 ms.

Время без omp - 1920.37 ms.

## Листинг

Компилятор - CLion 2021.3.3

C++ 20

**main.cpp**

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

```cpp
#include <omp.h>
#include <chrono>

// Сортировка подсчетом
int* sortR (int maxVal, const int* data, size_t size) {
    int* answer = new int[maxVal];
    for (int i = 0; i < maxVal; i++) {
        answer[i] = 0;
    }
#pragma omp parallel for default(none) schedule(static) shared(size, answer, data)
    for (size_t i = 0; i < size; i++) {
        answer[data[i]]++;
    }
    return answer;
}

int main(int argc, char* argv[]){
    std::string inFileName; // Имя входного файла
    std::string outFileName; // имя выходного файла
    int countThreads; // количество тредов
    double coefficient; // игнорируемая доля
    std::ifstream fin;
    std::string magicNumber;
    int width;
    int height;
    int maxVal;
    std::string maxValStr;
```

```
    if (argc != 5) {
            std::cout << "You must enter your input in format: <countThreads> "
                         "<input file name> <output file name> <coefficient>" << std::endl;
          std::cout << "input file format must be PPM or PGM" << std::endl;
          std::cout << "coefficient must be in range[0.0; 0.5)" << std::endl;
          std::cout << "countThreads must be an integer >= 0" << std::endl;
        return 1;
    }

    try {
        countThreads = std::stoi(argv[1]);
        if (countThreads < 0) {
                std::cout << "incorrect number of treads (< 0)" << std::endl;
            return 1;
        }
    } catch (const std::invalid_argument& e) {
          std::cout << "you write incorrect number of treads" << std::endl;
        return 1;
    }

    if (countThreads != 0) {
        omp_set_num_threads(countThreads);
    }
```

```
    inFileName = argv[2];
    outFileName = argv[3];

    try {
        coefficient = std::stod(argv[4]);
        if (coefficient < 0.0 || coefficient >= 0.5) {
             std::cout << "you wrote incorrect coefficient (must be 0
<= coefficient < 0.5)" << std::endl;
            return 1;
        }
    } catch (const std::invalid_argument& e) {
        std::cout << "" << std::endl;
        return 1;
    }

    std::string str;
    fin.open(inFileName, std::ios::in | std::ios::binary);
    if (!fin.is_open()) {
        std::cout << "This input file doesn't exist";
        return 1;
    }
    getline(fin, magicNumber);
    getline(fin, str);
    getline(fin, maxValStr);
    try {
        maxVal = stoi(maxValStr);
    } catch (const std::invalid_argument& e) {
        std::cout << "you write incorrect max value" << std::endl;
        return 1;
```

```
    }


    if (maxVal != 255) {
            std::cout << "Incorrect file (max value != 255)" <<
std::endl;
        return 1;
    }


//    запарсили width и height из строки
    int* str2 = new int[2];
    std::string str1 = str + " ";
    int beg = 0;
    int end = static_cast<int>(str1.find(' '));
    for (int i = 0; i < 2; i++) {
        try {
            str2[i] = stoi(str.substr(beg, end - beg));
        } catch (const std::invalid_argument& e) {
              std::cout << "you write incorrect width or height" <<
std::endl;
            return 1;
        }
        beg = end + 1;
        end = static_cast<int>(str1.find(' ', i + 1));
    }


    width = str2[0];
    height = str2[1];
    size_t size = height * width; // количество пикселей
       int needIgn = static_cast<int>(static_cast<double>(size) *
coefficient); // количество пикселей,
```

```
    // которых нужно проигнорировать с каждой стороны

    if (magicNumber == "P5") {

        // pgm

        int *data = new int[size];
         int downVal = 0; // первое значение, которые мы не должны игнорировать
          int upVal = maxVal; // последнее значение, которые мы не должны игнорировать

//          запись данных о пикселях в память
        for (size_t i = 0; i < size; i++) {
            data[i] = static_cast<int>(fin.get());
        }
        fin.close();

        auto start = std::chrono::high_resolution_clock::now();

        // сортируем значение пикселей
        int *values = sortR(maxVal + 1, data, size);
        int i;
        int minCol;
        int maxCol;

#pragma omp parallel sections default(none) private(i) shared(needIgn, values, downVal, upVal, minCol, maxCol)
        {
```

```
// находим минимальное значение пикселей, после учёта коэффициента игнорирования
#pragma omp section
            {
                i = needIgn;
                while (true) {
                    i -= values[downVal];
                    if (i < 0) {
                        break;
                    } else {
                        downVal++;
                    }
                }
                minCol = downVal;
            }

// находим максимальное значение пикселей, после учёта коэффициента игнорирования
#pragma omp section
            {
                i = needIgn;
                while (true) {
                    i -= values[upVal];
                    if (i < 0) {
                        break;
                    } else {
                        upVal--;
                    }
                }
```

```
                maxCol = upVal;
            }
        }
        if (maxCol != minCol) { // если картинка не одноцветная
// изменяем контрастность изображения, так,
// чтобы равномерно все растянуть (также учитываем, что у нас арифметика с насыщением)
#pragma omp parallel for default(none) schedule(static) shared(data, maxVal, minCol, maxCol, size)
            for (size_t j = 0; j < size; j++) {
                    data[j] = std::max(std::min(((data[j] - minCol) * maxVal) / (maxCol - minCol), maxVal), 0);
            }
        }

        auto finish = std::chrono::high_resolution_clock::now();
                                        auto      microseconds      = std::chrono::duration_cast<std::chrono::microseconds>(finish-start);

//         выводим данные в выходной файл
        std::ofstream out;
            out.open(outFileName, std::ios::out | std::ios::binary | std::ios::trunc);
        if (!out.is_open()) {
            std::cout << "This output file can't be read or found";
            return 1;
        }
         out << magicNumber << std::endl << width << " " << height << std::endl << maxVal << std::endl;
        for (size_t j = 0; j < size; j++) {
            out << static_cast<char>((std::byte) data[j]);
```

```
        }
        out.close();

            printf("Time (%i thread(s)) : %g ms\n", countThreads,
static_cast<double>(microseconds.count()) / 1000);

        delete[] data;
        delete[] values;


    } else if (magicNumber == "P6") {

//         ppm

//          все данные считываем также как и в pgm (только 3 цвета,
вместо 1)
        int* dataR = new int[size];
        int* dataG = new int[size];
        int* dataB = new int[size];
        int downValR = 0;
        int upValR = maxVal;
        int downValG = 0;
        int upValG = maxVal;
        int downValB = 0;
        int upValB = maxVal;
        for (int i = 0; i < size; i++) {
            dataR[i] = static_cast<int>(fin.get());
            dataG[i] = static_cast<int>(fin.get());
            dataB[i] = static_cast<int>(fin.get());
```

```
        }
        fin.close();

        auto start = std::chrono::high_resolution_clock::now();

        int i;
        // сортируем значение пикселей
        int* valuesR = sortR(maxVal + 1, dataR, size);
        int* valuesG = sortR(maxVal + 1, dataG, size);
        int* valuesB = sortR(maxVal + 1, dataB, size);

// находим минимальные и максимальные значения пикселей, после учёта коэффициента игнорирования
#pragma omp parallel sections default(none) private(i) shared(needIgn, valuesR, valuesG, valuesB, dataR, dataG, dataB, size, downValR, downValG, downValB, upValR, upValG, upValB, maxVal)
        {
#pragma omp section
            {
                i = needIgn;
                while (true) {
                    i -= valuesR[downValR];
                    if (i < 0) {
                        break;
                    } else {
                        downValR++;
                    }
                }
            }
#pragma omp section
```

```
            {
                i = needIgn;
                while (true) {
                    i -= valuesG[downValG];
                    if (i < 0) {
                        break;
                    } else {
                        downValG++;
                    }
                }
            }
#pragma omp section
            {
                i = needIgn;
                while (true) {
                    i -= valuesB[downValB];
                    if (i < 0) {
                        break;
                    } else {
                        downValB++;
                    }
                }
            }
#pragma omp section
            {
                i = needIgn;
                while (true) {
                    i -= valuesR[upValR];
                    if (i < 0) {
```

```
                    break;
                } else {
                    upValR--;
                }
            }
        }
#pragma omp section
        {
            i = needIgn;
            while (true) {
                i -= valuesG[upValG];
                if (i < 0) {
                    break;
                } else {
                    upValG--;
                }
            }
        }
#pragma omp section
        {
            i = needIgn;
            while (true) {
                i -= valuesB[upValB];
                if (i < 0) {
                    break;
                } else {
                    upValB--;
                }
            }
```

```
            }
        }

            // находим минимальное и максимальное значение из всех цветов
                int minCol = std::min(downValR, std::min(downValG, downValB));
        int maxCol = std::max(upValR, std::max(upValG, upValB));

        if (minCol != maxCol) { // если картинка не одноцветная

// изменяем контрастность изображения, так,
// чтобы равномерно все растянуть (также учитываем, что у нас арифметика с насыщением)
#pragma omp parallel for default(none) schedule(static) shared(dataR, dataG, dataB, maxVal, minCol, maxCol, size)
            for (size_t j = 0; j < size; j++) {
                dataR[j] = std::max(std::min(((dataR[j] - minCol) * maxVal) / (maxCol - minCol), maxVal), 0);
                dataG[j] = std::max(std::min(((dataG[j] - minCol) * maxVal) / (maxCol - minCol), maxVal), 0);
                dataB[j] = std::max(std::min(((dataB[j] - minCol) * maxVal) / (maxCol - minCol), maxVal), 0);
            }
        }

        auto finish = std::chrono::high_resolution_clock::now();
                                    auto microseconds = std::chrono::duration_cast<std::chrono::microseconds>(finish-start);

// выводим данные в выходной файл
```

```
        std::ofstream out;
            out.open(outFileName, std::ios::out | std::ios::binary |
std::ios::trunc);
        if (!out.is_open()) {
            std::cout << "This output file can't be read or found";
            return 1;
        }
         out << magicNumber << std::endl << width << " " << height <<
std::endl << maxVal << std::endl;
        for (size_t  j = 0; j < size; j++) {
            out << static_cast<char>((std::byte) dataR[j])
                << static_cast<char>((std::byte) dataG[j])
                << static_cast<char>((std::byte) dataB[j]);
        }
        out.close();

             printf("Time (%i thread(s)) : %g ms\n", countThreads,
static_cast<double>(microseconds.count()) / 1000);

        delete[] dataR;
        delete[] dataG;
        delete[] dataB;

        delete[] valuesR;
        delete[] valuesG;
        delete[] valuesB;
    } else {
        std::cout << "Wrong object (not PGM or PPM)";
    }
}
```