Лабораторная работа №4	M3139	2023
OpenMP	Гоге Анастасия Эдуардовна	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: язык программирования C++, компилятор g++ 12.2.0, стандарт OpenMP 2.0.

Вариант: hard.

1. **Описание конструкций OpenMP для распараллеливания команд.**

#pragma omp parallel

С помощью этой конструкции объявляется блок, в котором будет происходить распараллеливание.

#pragma omp for

Эта конструкция распределяет итерации цикла for по потокам. К ней можно приписать nowait и тогда потоки не будут ждать выполнения остальных, чтобы продолжить работу.

schedule(тип, размер блока) – определяет, как будут распределены итерации по потокам.

Типы:

static – итерации равномерно распределяются по потокам. Если при этом задан размер блока, то все итерации блоками заданного размера поочередно распределяются между потоками.

dynamic – итерации распределяется блоками заданного размера между потоками (по умолчанию размер равен 1). Как только поток заканчивает обработку своего блока, он получает следующий.

Static может работать медленнее, чем dynamic, если в разных итерациях цикла выполняет разное количество операций. Но при этом в dynamic больше времени тратится на распределение блоков между потоками. По умолчанию установлен static и блоки делятся между потоками примерно поровну.

Также есть другие типы (guided, runtime), но я их не использую.

omp_set_num_threads(количество тредов) – устанавливает количество потоков, которое может быть использовано для исполнения блоков.

#pragma omp critical

Это критическая секция, она выполняется одним потоком в определенный момент времени, а остальные потоки ждут, когда выполнение другим потоком закончится.

2. Описание работы написанного кода.

Программа начинается с проверки заданных аргументов. Они вводятся так: args[1] – количество потоков, args[2] – название входного файла, args[3] – названия выходного файла.

Далее открываем файл (листинг 1), указатель ставим на конец, вычисляем размер файла и указатель возвращаем в начало. С помощью команды read весь файл считывается в массив.

```
try {
    ifstream file(args[2], ios_base::binary);

    file.seekg(0, ifstream::end);
    length = file.tellg();
    file.seekg(0, ifstream::beg);

    bytes = new uint8_t[length];

    file.read((char*) bytes, length);
    file.close();
} catch (...) {
    cout << "input error";
    return 0;
}
```

Листинг 1. Считывание файла

Потом проверяем правильность формата файла и вычисляется размер картинки.

В зависимости от количества тредов выполняется или часть кода без OpenMP, если оно равно -1, или часть с OpenMP. Опишем часть кода с OpenMP, вторая часть работает также, только без использования OpenMP.

Сначала задаем количество тредов или, если оно равно нулю, то не задаем и оставляем по умолчанию. Далее в цикле for анализ файла проделывается нужное количество раз (изначально указано значение 1, можно поменять значение переменной num_of_runs, чтобы запустить нужное количество раз).

В массиве brights[256] будут храниться количества пикселей каждой яркости (гистограмма). Заполняем его нулями, этот цикл нет смысла распараллеливать, так как он маленький.

Чтобы параллельно вычислять гистограмму (листинг 2) создадим для каждого потока свой массив brights_i[256], заполим их и потом в critical сложим, чтобы посчиталось правильно.

```
#pragma omp parallel
            {
                int brights_i[256];
                for (int & j : brights_i) {
                    j = 0;
                }
#pragma omp for nowait
                for (int i = idx; i < length; i++) {
```

```
                    brights_i[bytes[i]]++;
                }

#pragma omp critical
                for (int j = 0; j < 256; j++) {
                    brights[j] += brights_i[j];
                }
            }
```

Листинг 2. Вычисление гистограммы.

Далее будем вычислять межкластерную дисперсию для всех наборов порогов (σ в документе с описание алгоритма, d в коде). Это делается по формуле $\sigma^2 = \sum_{k=1}^{4} q_k \mu_k^2 - \mu^2$, но поскольку μ (среднее значение яркости всего изображения) не меняется, максимизируем $\sigma^2 = \sum_{k=1}^{4} q_k \mu_k^2$, где q – это вероятность кластера, она равна сумме по всем вероятностям встречаемости яркостей, лежащих в кластере, а μ – среднее значение кластера, равное сумме $\frac{fp(f)}{q_k}$ по всем яркостям кластера(f – количество пикселей данной яркости). Чтобы посчитать это быстро, предподсчитаем префиксные суммы вероятностей (точнее просто частот, так как мы не делим на количество, чтобы делать это быстрее) и средних значений, не деленных на вероятность (листинг 3).

```
long long pref_p[256];
pref_p[0] = brights[0];
for (int i = 1; i < 256; i++) {
    pref_p[i] = pref_p[i - 1] + brights[i];
}

long long pref_m[256];
pref_m[0] = 0;
for (int i = 1; i < 256; i++) {
    pref_m[i] = pref_m[i - 1] + brights[i] * i;
}
```

Листинг 3. Префиксные суммы.

Теперь мы можем вычислить межкластерную дисперсию для фиксированного набора за O(1)(листинг 4). $m_k = \mu_k q_k$, прибавить нужно $q_k \mu_k^2 = \frac{m_k^2}{q_k}$. При таком вычислении получается дисперсия, умноженная на количество элементов в файле, но это не мешает максимизировать, поэтому просто не будем делить.

Чтобы ускорить этот перебор, можно распараллелить один из циклов for. Если разбивать внешний for, то по потокам будут распределяться блоки, содержащие большое количество операций, а если внутренний, то маленькое, и поэтому в таком случае получится разделить равномернее, но придется распределять большее количество блоков, для этого понадобится время. Экспериментально было выяснено, что с распараллеливанием внутреннего цикла получается быстрее. В critical сравниваем все получившиеся максимумы d.

```
#pragma omp parallel
            {
                double max_d_i = 0;
                int fi[3];

                double d;
                long long m1;
                long long m2;
                long long m3;
                long long m4;

                for (int f0 = 0; f0 < L - M + 1; f0++) {
                    for (int f1 = f0 + 1; f1 < L - M + 2; f1++) {
#pragma omp for nowait
                        for (int f2 = f1 + 1; f2 < L - M + 3; f2++) {
                            m1 = pref_m[f0];
                            m2 = pref_m[f1] - pref_m[f0];
                            m3 = pref_m[f2] - pref_m[f1];
                            m4 = pref_m[L - 1] - pref_m[f2];

                            d = 0;
                            d += (double) (m1 * m1) / (double) pref_p[f0];
                            d += (double) (m2 * m2) / (double) (pref_p[f1]
                            - pref_p[f0]);
                            d += (double) (m3 * m3) / (double) (pref_p[f2]
                            - pref_p[f1]);
                            d += (double) (m4 * m4) / (double)
                            (pref_p[L - 1] - pref_p[f2]);

                            if (d > max_d_i) {
                                max_d_i = d;
                                fi[0] = f0;
                                fi[1] = f1;
                                fi[2] = f2;
                            }
                        }
                    }
                }

#pragma omp critical
                if (max_d_i > max_d) {
                    max_d = max_d_i;
                    f[0] = fi[0];
                    f[1] = fi[1];
                    f[2] = fi[2];
                }

            }
```

Листинг 4. Вычисление дисперсии.

Когда пороги вычислены, параллельно перезаписываем массив с файлом (листинг 5), если это последний запуск, иначе просто проходим по циклу, чтобы время посчиталось. Создать еще один массив нельзя, так как не

гарантированно, что в память поместятся два изображения. Далее записываем это в файл с помощью команды write и считаем время.

```
#pragma omp parallel
            {
#pragma omp for
                for (int i = idx; i < length; i++) {
                    if (run_num == num_of_runs) {
                        if (bytes[i] <= f[0]) {
                            bytes[i] = 0;
                        } else if (bytes[i] <= f[1]) {
                            bytes[i] = 84;
                        } else if (bytes[i] <= f[2]) {
                            bytes[i] = 170;
                        } else {
                            bytes[i] = 255;
                        }
                    }
                }

            }

try {
    ofstream file2(args[3], ios_base::binary);
    file2.write((char*) bytes, length);
    file2.close();
} catch (...) {
    cout << "output error";
    return 0;
}
```

Листинг 5. Запись нового изображения.

В конце выводим среднее время по всем запускам.

3. Результат работы написанной программы.

Процессор: AMD Ryzen 5 4500U with Radeon Graphics 2.38 GHz

Вывод в консоль при запуске с дефолтным количеством тредов и дефолтным schedule:

```
Time (0 thread(s)): 18.27 ms
77 130 187
```

Получившееся изображение:

Рисунок 1. Получившееся изображение.

4. Экспериментальная часть.

На рисунке 2 представлен график зависимости времени работы программы (в ms) от количества потоков при типе schedule – static без указания размера блока (первые два результата, обозначенные на горизонтальной оси -1 и 0 – это запуск без OpenMP и без указания количества потоков, то есть 8 в моем случае). От одного потока до восьми (то есть до максимума, который может быть на моем компьютере) программа ускоряется, а потом понемногу замедляется.

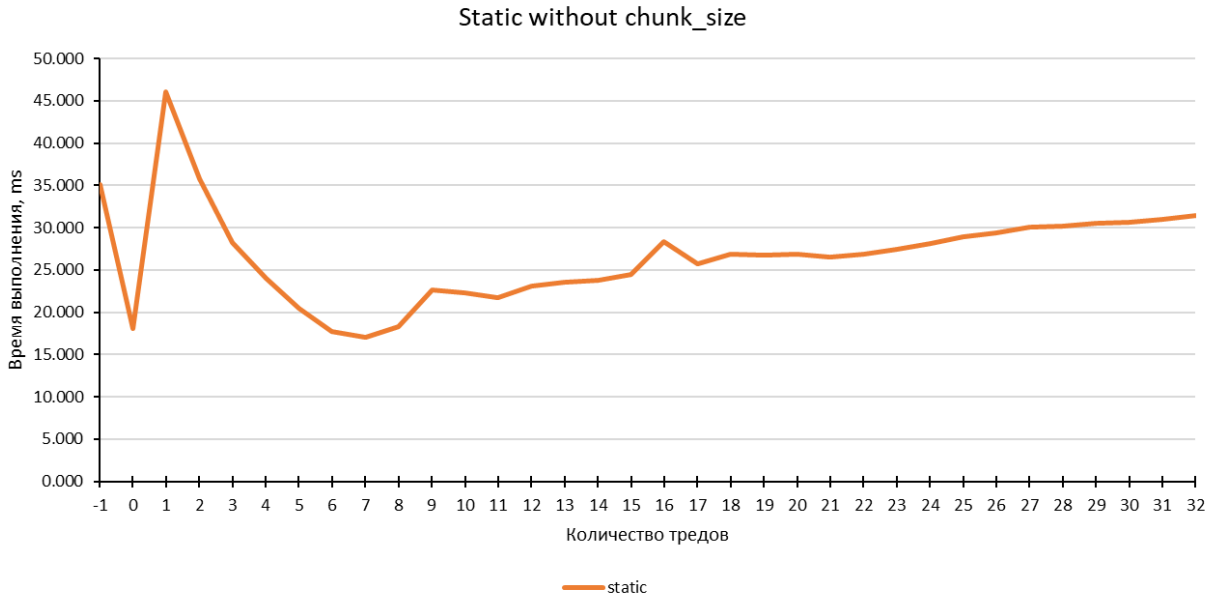

Рисунок 2. График зависимости времени работы программы (в ms) от количества тредов (static).

На рисунке 3 представлен график зависимости времени работы программы (в ms) от количества потоков при типе schedule – dynamic без указания размера блока (первые два результата, обозначенные на горизонтальной оси -1 и 0 – это запуск без OpenMP и без указания количества потоков, то есть 8 в моем случае). Можно заметить, что до 8 тредов время ументшается, а потом с увеличением числа потоков увеличивается, такая ситуация наблюдается из-за того, что много времени тратится на распределение блоков между потоками.

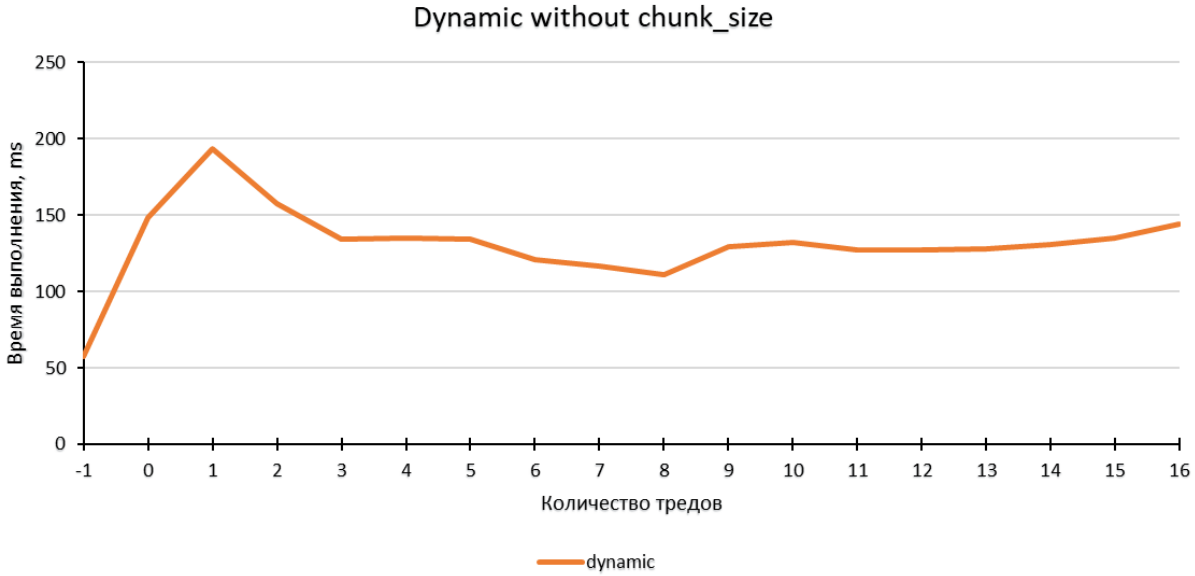

Рисунок 3. График зависимости времени работы программы (в ms) от количества потоков (dynamic).

На рисунке 4 представлен график зависимости времени работы программы (в ms) от размера блока в schedule при типе schedule – static. Разные графики показывают результаты при разном количестве потоков. Время выполнения с одним тредом почти не меняется, потому что блоки все равно выполняет один тред. А в остальных случаях с увеличением размера блока время уменьшается, потому что маленькие блоки можно поделить более эффективно между потоками, так чтобы каждый тред выполнял примерно поровну итераций.

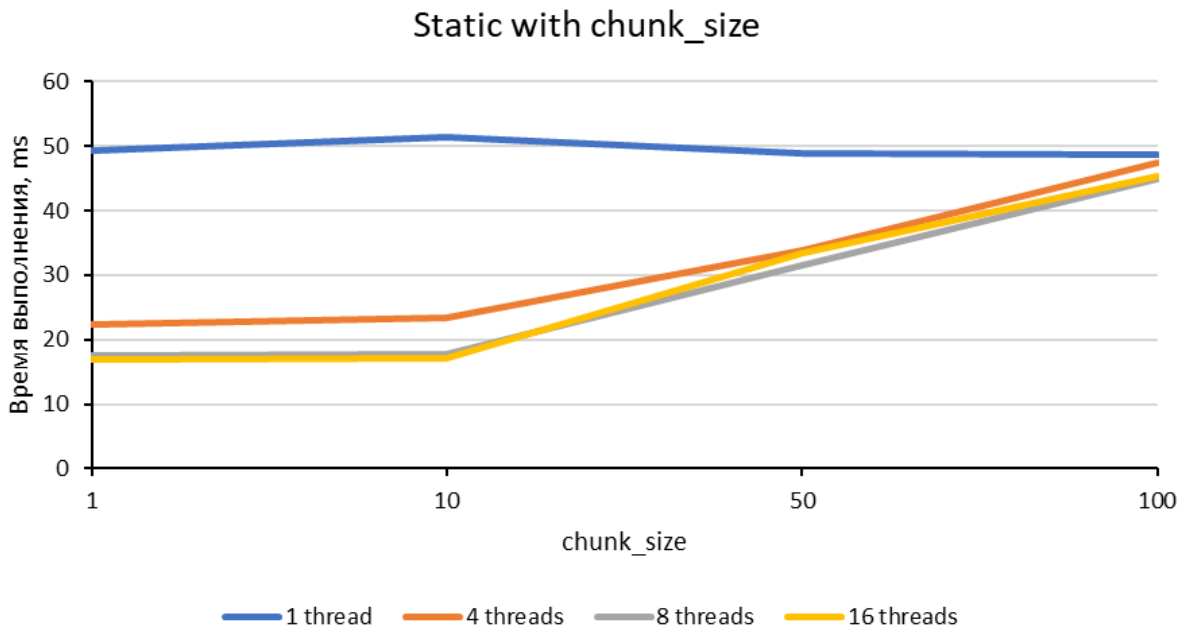

Рисунок 4. График зависимости времени работы программы (в ms) от размера блока в schedule при типе schedule – static

На рисунке 5 представлен график зависимости времени работы программы (в ms) от размера блока в schedule при типе schedule – dynamic. Разные графики показывают результаты при разном количестве потоков. Здесь можно наблюдать ситуацию похожую на рисунок 4 по тем же причинам.

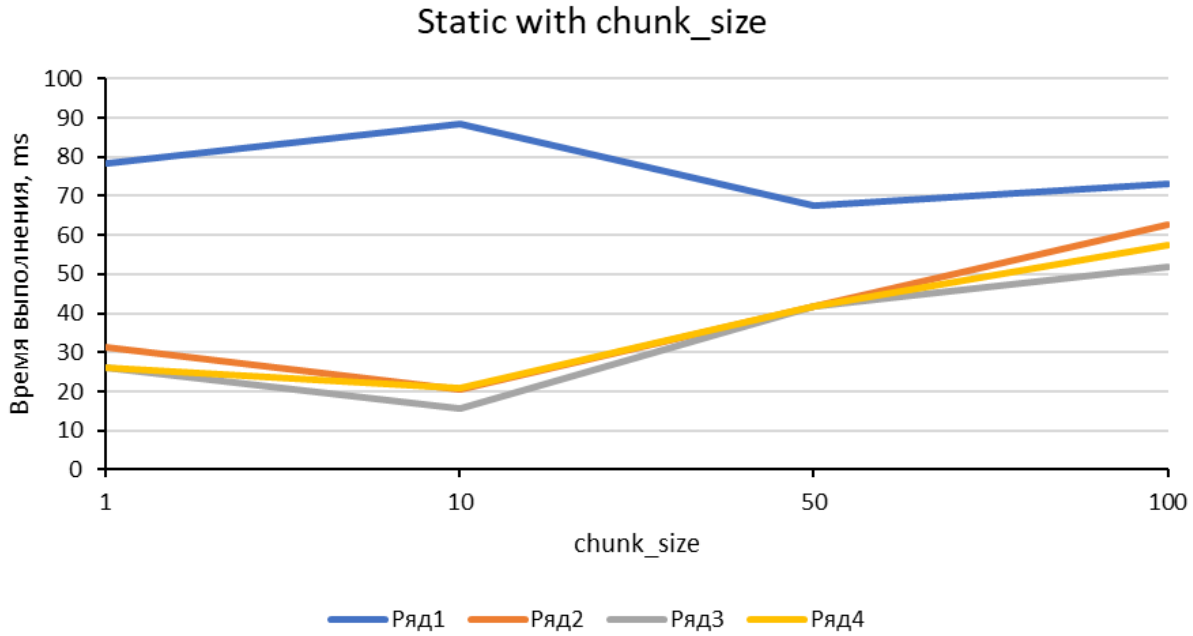

Рисунок 5. График зависимости времени работы программы (в ms) от размера блока в schedule при типе schedule – dynamic.

Что касается сравнения скорости выполнения без OpenMP и с одним потоком, его можно увидеть на рисунках 2 и 3. Без OpenMP выполняется быстрее (35 ms), чем с одним потоком и в static (46 ms), и в dynamic (193 ms).

5. Список источников.

1. https://www.openmp.org//wp-content/uploads/cspec20.pdf - OpenMP C and C++ Application Program Interface

6. Листинг кода.

hard.cpp

```cpp
#include <iostream>
#include <omp.h>
#include <fstream>
#include <cassert>
#include <string>

using namespace std;

const int L = 256;
const int M = 4;

int main(int num_of_arguments, char *args[]) {
    assert("not 4 arguments" && num_of_arguments == 4);
    int num_of_threads;
    try {
        num_of_threads = stoi(args[1]);
    } catch (...) {
        cout << "number of threads is not a number";
        return 0;
    }
    assert("number of threads smaller then -1" && num_of_threads > -2);

    uint8_t *bytes;
    long long length;

    try {
        ifstream file(args[2], ios_base::binary);

        file.seekg(0, ifstream::end);
        length = file.tellg();
        file.seekg(0, ifstream::beg);

        bytes = new uint8_t[length];

        file.read((char*) bytes, length);
        file.close();
    } catch (...) {
        cout << "input error";
        return 0;
    }

    string type;
    type += (char) bytes[0];
    type += (char) bytes[1];
    type += (char) bytes[2];
    assert("no P5" && type == "P5\n");

    string width;
    int idx = 3;
    while (bytes[idx] != ' ') {
        assert("not number" && bytes[idx] >= '0' && bytes[idx] <= '9');
```

```
        width += (char) bytes[idx];
        idx++;
    }

    idx++;
    string height;
    while (bytes[idx] != '\n') {
        assert("not number" && bytes[idx] >= '0' && bytes[idx] <= '9');
        height += (char) bytes[idx];
        idx++;
    }

    idx++;
    string bright;
    while (bytes[idx] != '\n') {
        bright += (char) bytes[idx];
        idx++;
    }
    assert("not 255" && bright == "255");

    idx++;
    double time = 0;
    int f[3];

    int num_of_runs = 100;
    if (num_of_threads == -1) {

        for (int run_num = 0; run_num <= num_of_runs; run_num++) {
            double t = omp_get_wtime();

            int brights[256];
            for (int & i : brights) {
                i = 0;
            }

            for (int i = idx; i < length; i++) {
                brights[bytes[i]]++;
            }

            long long pref_p[256];
            pref_p[0] = brights[0];
            for (int i = 1; i < 256; i++) {
                pref_p[i] = pref_p[i - 1] + brights[i];
            }

            long long pref_m[256];
            pref_m[0] = 0;
            for (int i = 1; i < 256; i++) {
                pref_m[i] = pref_m[i - 1] + brights[i] * i;
            }

            double max_d = 0;

            double d;
            long long m1;
            long long m2;
            long long m3;
            long long m4;

            for (int f0 = 0; f0 < L - M + 1; f0++) {
                for (int f1 = f0 + 1; f1 < L - M + 2; f1++) {
                    for (int f2 = f1 + 1; f2 < L - M + 3; f2++) {
```

```
                        d = 0;
                        m1 = pref_m[f0];
                        d += (double) (m1 * m1) / (double) pref_p[f0];
                        m2 = pref_m[f1] - pref_m[f0];
                        d += (double) (m2 * m2) / (double) (pref_p[f1] -
pref_p[f0]);
                        m3 = pref_m[f2] - pref_m[f1];
                        d += (double) (m3 * m3) / (double) (pref_p[f2] -
pref_p[f1]);
                        m4 = pref_m[L - 1] - pref_m[f2];
                        d += (double) (m4 * m4) / (double) (pref_p[L - 1] -
pref_p[f2]);

                        if (d > max_d) {
                            max_d = d;
                            f[0] = f0;
                            f[1] = f1;
                            f[2] = f2;
                        }
                    }
                }
            }

            for (int i = idx; i < length; i++) {
                if (run_num == num_of_runs) {
                    if (bytes[i] <= f[0]) {
                        bytes[i] = 0;
                    } else if (bytes[i] <= f[1]) {
                        bytes[i] = 84;
                    } else if (bytes[i] <= f[2]) {
                        bytes[i] = 170;
                    } else {
                        bytes[i] = 255;
                    }
                }
            }

            try {
                ofstream file2(args[3], ios_base::binary);
                file2.write((char*) bytes, length);
                file2.close();
            } catch (...) {
                cout << "output error";
                return 0;
            }

            double t1 = omp_get_wtime();
            time += (t1 - t);
        }

    } else {

        if (num_of_threads != 0) {
            omp_set_num_threads(num_of_threads);
        }

        for (int run_num = 1; run_num <= num_of_runs; run_num++) {
            double t = omp_get_wtime();

            int brights[256];
            for (int & i : brights) {
                i = 0;
```

```
            }

#pragma omp parallel
            {
                int brights_i[256];
                for (int & j : brights_i) {
                    j = 0;
                }
#pragma omp for nowait
                for (int i = idx; i < length; i++) {
                    brights_i[bytes[i]]++;
                }

#pragma omp critical
                for (int j = 0; j < 256; j++) {
                    brights[j] += brights_i[j];
                }
            }

            long long pref_p[256];
            pref_p[0] = brights[0];
            for (int i = 1; i < 256; i++) {
                pref_p[i] = pref_p[i - 1] + brights[i];
            }

            long long pref_m[256];
            pref_m[0] = 0;
            for (int i = 1; i < 256; i++) {
                pref_m[i] = pref_m[i - 1] + brights[i] * i;
            }

            double max_d = 0;


#pragma omp parallel
            {
                double max_d_i = 0;
                int fi[3];

                double d;
                long long m1;
                long long m2;
                long long m3;
                long long m4;


                for (int f0 = 0; f0 < L - M + 1; f0++) {
                    for (int f1 = f0 + 1; f1 < L - M + 2; f1++) {
#pragma omp for nowait
                        for (int f2 = f1 + 1; f2 < L - M + 3; f2++) {
                            m1 = pref_m[f0];
                            m2 = pref_m[f1] - pref_m[f0];
                            m3 = pref_m[f2] - pref_m[f1];
                            m4 = pref_m[L - 1] - pref_m[f2];

                            d = 0;
                            d += (double) (m1 * m1) / (double) pref_p[f0];
                            d += (double) (m2 * m2) / (double) (pref_p[f1] -
pref_p[f0]);
                            d += (double) (m3 * m3) / (double) (pref_p[f2] -
pref_p[f1]);
                            d += (double) (m4 * m4) / (double) (pref_p[L - 1] -
```

```
pref_p[f2]);

                            if (d > max_d_i) {
                                max_d_i = d;
                                fi[0] = f0;
                                fi[1] = f1;
                                fi[2] = f2;
                            }
                        }
                    }
                }

#pragma omp critical
                if (max_d_i > max_d) {
                    max_d = max_d_i;
                    f[0] = fi[0];
                    f[1] = fi[1];
                    f[2] = fi[2];
                }

            }

#pragma omp parallel
            {
#pragma omp for
                for (int i = idx; i < length; i++) {
                    if (run_num == num_of_runs) {
                        if (bytes[i] <= f[0]) {
                            bytes[i] = 0;
                        } else if (bytes[i] <= f[1]) {
                            bytes[i] = 84;
                        } else if (bytes[i] <= f[2]) {
                            bytes[i] = 170;
                        } else {
                            bytes[i] = 255;
                        }
                    }
                }

            }

            try {
                ofstream file2(args[3], ios_base::binary);
                file2.write((char*) bytes, length);
                file2.close();
            } catch (...) {
                cout << "output error";
                return 0;
            }
            double t1 = omp_get_wtime();
            time += (t1 - t);
        }
    }
    printf("Time (%i thread(s)): %g ms\n", num_of_threads, time * 1000 / (float)
num_of_runs);
    printf("%u %u %u\n", f[0], f[1], f[2]);
    delete[] bytes;
    return 0;
}
```

Листинг 6. Весь код