САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил: Андосов Герман Андреевич

Номер ИСУ: 334875

студ. гр. М3139

Санкт-Петербург

2021

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

По ходу развития вычислительной техники люди искали варианты ускорения компьютера. Поначалу для увеличения вычислительной мощности стали появляться многопроцессорные системы. Идея была очень простой – если несколько копий одной и той же программы можно запустить сразу на нескольких процессорах и это будет иметь смысл, то за единицу времени компьютер сможет, очевидно, произвести больше вычислений. Но вскоре появились многоядерные процессоры, которые сменили многопроцессорные системы. Их преимущество заключалось в том, что ядра одного процессора расположены близко друг к другу, и не требовалось проводить высокоскоростную шину между несколькими процессорами.

Таким образом, большая часть современных процессоров теперь состоит из нескольких вычислительных блоков – ядер. А вместо понятия процесса, который исполняется процессором, появилось понятие треда – исполнительной части процесса, которая исполняется ядром.

Не секрет, что основное время работы программ составляют циклы. Если программа исполняет какой-либо цикл, и все итерации независимы друг от друга, то такой процесс цикла можно разбить на несколько тредов и исполнить эти треды одновременно параллельно (в дальнейшем я буду употреблять выражение «распараллелить цикл»). Тогда в работе будет

задействовано не одно ядро процессора, а сразу несколько, что даст нам многократный прирост по скорости выполнения этого цикла.

Все попытки сделать автоматическое разбиение программы на треды (аппаратно или на уровне компиляции) были неудачными, поэтому и по сей день треды создаются программистами.

Создать треды можно, например, с помощью OpenMP, о котором будет рассказано далее. Но перед тем, как что-то распараллелить, программист должен понимать и учитывать множество различных нюансов:

1) Создание треда – дорогая операция, поэтому все треды должны быть достаточно «тяжелыми», то есть они должны содержать относительно большое количество операций.
2) Количество операций в тредах должно быть примерно одинаковым. Действительно, если, например, один тред будет заметно тяжелее других, то все остальные треды будут простаивать и ждать, пока исполнится самый тяжелый тред. Если ядра простаивают, то это означает потерю времени.
3) Треды могут выполняться как одновременно, так и не одновременно. Рассмотрим простой пример. Пусть у нас есть два треда, которые исполняют одну и ту же программу:

   $x = x + 1.$

   На самом деле, в этой программе скрыты 3 действия: чтение, арифметическое действие и запись. Если, например, чтение произойдет одновременно, то оба треда после арифметической операции запишут значение $x + 1$. Хотя мы, конечно, ожидали, что

значение изменится на 2, а не на 1. Подобные ситуации имеют название – состояние гонки, и их нужно избегать.

Еще один важный момент – количество тредов может превышать количество ядер процессора и это нормально. В таком случае освободившиеся ядра будут забирать себе на исполнение треды из очереди.

Теперь перейдем к OpenMP. Это специальный программный интерфейс (API), предназначенный для создания тредов и написания многопоточных программ. Для того, чтобы использовать OpenMP в программе на C++, достаточно написать одну строчку, подключающую файл omp.h.

Для начала разберемся, как указать, на сколько тредов будут разбиваться куски нашего кода. Для этого есть функция omp_set_num_threads(x), где x – это желаемое количество тредов.

Для создания треда внутри программы существуют так называемые директивы. Директивы OpenMP имеют вид:

```
#pragma omp /* название */ /* клозы */
```

Клозы – это аргументы или настройки для директивы.

Одной из главных директив является parallel:

```
#pragma omp parallel default(none) shared(/* переменные */)
{


}
```

Весь код, написанный внутри фигурных скобок, будет исполнен параллельно каждым тредом. Внешние переменные, объявленные ранее,

будут невидимы для кода внутри фигурных скобок. Все нужные нам переменные нужно будет указать в клозе shared. Но внутри этого блока можно, например, создавать переменные, которые будут доступны только одному треду.

Но основная наша цель – распараллелить цикл. Приведу пример того, как не стоит это делать:

```
#pragma omp parallel default(none) shared(/* переменные */)
{
   for (...) {

   }
}
```

Не стоит забывать, что код внутри фигурных скобок, принадлежащих директиве, будет исполнен каждым тредом. То есть, написанный цикл for, наоборот, будет считать одно и то же несколько раз! Для того, чтобы распараллелить цикл, нужна еще одна директива – for:

```
#pragma omp parallel default(none) shared(/* переменные */)
{
   #pragma omp for schedule(/* вид schedule */)
   for (...) {

   }
}
```

Теперь мы сообщили программе, что итерации цикла должны быть распределены между тредами. Как именно распределены – для этого есть

клоз schedule, в котором можно это настроить. Для предстоящей задачи нам понадобятся два:

1) static(chunk_size) – разбивает цикл на блоки размером chunk_size итераций. Назначение блоков тредам происходит статически (одномоментно), причем блоки назначаются тредам по круговой системе (первый блок – первому треду, второй – второму, …, x-й – последнему, $(x + 1)$-й – первому и т.д.). Если chunk_size не указан явно, то он автоматически будет примерно равен количеству итераций, деленному на количество тредов.
2) dynamic(chunk_size) – разбивает цикл на блоки размером chunk_size итераций. Назначение блоков тредам происходит динамически (по ходу выполнения программы). Если chunk_size не указан явно, то он автоматически будет равен единице.

Кроме того, OpenMP умеет параллелить только циклы, написанные в так называемой канонической форме. Определение канонической формы цикла for довольно долгое, и, чтобы не захламлять отчет не самой важной информацией, я просто приведу часть документации OpenMP, где написано, что это такое (см. рис. 1).

for (*init-expr*; *var* *logical-op* *b*; *incr-expr*)	
init-expr	One of the following: *var* = *lb* *integer-type var* = *lb*
incr-expr	One of the following: ++*var* *var*++ --*var* *var*-- *var* += *incr* *var* -= *incr* *var* = *var* + *incr* *var* = *incr* + *var* *var* = *var* - *incr*
var	A signed integer variable. If this variable would otherwise be shared, it is implicitly made private for the duration of the **for**. This variable must not be modified within the body of the **for** statement. Unless the variable is specified **lastprivate**, its value after the loop is indeterminate.
logical-op	One of the following: < <= > >=
lb, *b*, and *incr*	Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

Рисунок №1 – Строгое определение канонической формы цикла for.

Замечу, что несколько директив можно уместить на одной строчке, это может быть удобно:

```
#pragma omp parallel for default(none) schedule(static)

for (...) {


}
```

Рассмотрим еще одну важную директиву – critical:

```
#pragma omp parallel default(none) shared(/* переменные */)
{
   #pragma omp critical
   {
       // Code
   }
}
```

Код внутри фигурных скобок critical'а может быть исполнятся не более чем одним тредом в каждый момент времени.

Еще одна директива, которая мне пригодилась на практике – это single:

```
#pragma omp parallel default(none) shared(/* переменные */)
{
   #pragma omp single
   {
       // Code
   }
}
```

Она означает блок кода, который будет выполнен только одним тредом. Есть принципиальное различие между critical и single – в первом случае код будет исполнен всеми тредами, а во втором – только одним.

Еще одна полезная директива – barrier:

```
#pragma omp parallel default(none) shared(/* переменные */)
{
   // Код
```

```
    #pragma omp barrier

    // Продолжение кода
}
```

По большому счету она означает, что код, написанный выше этой директивы, должен исполниться всеми тредами, и только после этого будет исполнен код снизу.

Разумеется, использование трех последних директив бьет по времени работы программы, но иногда они необходимы.

Итак, для наглядной демонстрации работы с OpenMP, решим следующую задачу:

Изображение может иметь плохую контрастность: используется не весь диапазон значений, а только его часть. Например, если самые тёмные точки изображения имеют значение 20, а не 0.

Задание состоит в том, чтобы изменить значения пикселей таким образом, чтобы получить максимальную контрастность: растянуть диапазон значений до [0; 255], но при этом не изменить оттенки (то есть в цветных изображениях нужно одинаково изменять каналы R, G и B).

<параметры_алгоритма> = <коэффициент>

При вычислении растяжения игнорировать некоторую долю (по количеству) самых тёмных и самых светлых точек (для RGB в каждом канале отдельно): ***<коэффициент>*** *(диапазон значений [0.0, 0.5)). Это позволяет игнорировать шум, который незаметен глазу, но мешает автоматической настройке контрастности. Растяжение диапазона*

следует выполнять с насыщением, чтобы проигнорированные пиксели не вышли за границы [0; 255].

Моя программа будет уметь работать с изображениями формата PNM, а именно PGM (черно-белые изображения) и PPM (цветные изображения) (см. рис. 2):

P5 (PGM)	“P5\n<width> <height>\n255\n<Gray_data>”
P6 (PPM)	“P6\n<width> <height>\n255\n<RGB_data>”

Рисунок №2 – Поддерживаемые форматы файлов и их структура.

<Gray_data> и <RGB_data> – это наборы идущих подряд байтов. <Gray_data> это $height \cdot width$ байт, каждый из которых отвечает за один пиксель изображения. Порядок байтов привычный – самый первый байт – самый левый пиксель изображения, следующий байт – справа от него, и так далее. <RGB_data> по сути то же самое, но каждый пиксель характеризуется тремя байтами – каналами R, G и B соответственно, идущими подряд.

Опишу алгоритм, который решает эту задачу. После чтения заголовка файла и обработки ошибок программа разделяется на две большие части. Одна из них работает с черно-белыми изображениями, а другая – с цветными. Для определенности я буду описывать алгоритм для цветных изображений, поскольку алгоритм для черно-белых изображений абсолютно аналогичен и даже проще, поскольку нужно обрабатывать лишь один цветовой канал, а не три.

Для того, чтобы понять, какие цвета я буду игнорировать в зависимости от коэффициента, мне нужна будет статистика того, как часто встречается каждый из 256 цветов в каждом из каналов R, G, B. Когда у нас есть эта статистика, несложно посчитать границы отрезков цветов, которые не должны игнорироваться нашей программой. Итого у нас будет три левых границы и три правых границы, для каждого цвета. По условию задачи среди левых границ мы должны взять минимальную, а среди правых – максимальную. Теперь у нас есть некий отрезок (обозначим его границы как left_border и right_border), который нужно «растянуть» в отрезок от 0 до 255. Для лучшего понимания приведу картинку (см. рис. 3).

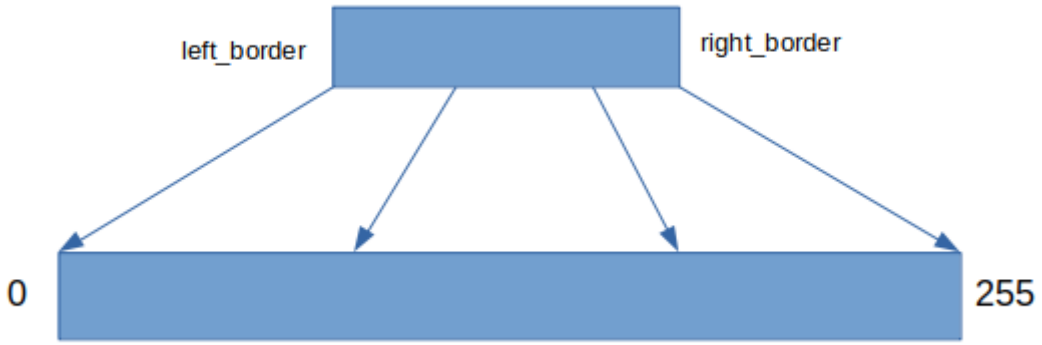

Рисунок №3 – Наглядный пример того, как растягивается диапазон значений пикселей.

Что делать с цветами, которые не попали в отрезок [left_border:right_border]? Об этом сказано в условии – цвета, меньшие чем left_border станут равны нулю, а большие чем right_border станут равны 255. Формула для растяжения попавших в отрезок значений очень проста:

$$newcol = \frac{col - leftborder}{rightborder - leftborder} \cdot 255$$

И вместо того, чтобы применять эту формулу к каждому пикселю изображения, я предпосчитаю для каждого из 256 цветов его будущее значение. Это позволить не делать массу одинаковых вычислений.

Дальше просто пробегаемся по картинке, ставим каждому пикселю новое значение и выводим полученный результат.

Практическая часть

Расскажу о том, как устроен мой код. Сначала аргументы командной строки проверяются на корректность. Если все хорошо, то с помощью команды omp_set_num_threads я устанавливаю количество тредов. Изображение я открываю с помощью ifstream, затем считываю четыре переменные, составляющие заголовок файла – тип (P5 или P6), ширину и высоту картинки в пикселях, а также максимальное значение цвета. Производится обработка ошибок. После этого моя программа разделяется на две большие части – одна работает с черно-белыми изображениями (PGM), а другая – с цветными (PPM). Как я говорил ранее, буду описывать алгоритм для цветных изображений.

Итак, я считываю значения R, G, B для всех пикселей. Теперь я хочу посчитать для каждого цветового канала – сколько раз каждое из 256 значений встретилось в файле. Для этого я завожу три массива размером 256, заполненных нулями: freq_r, freq_g, freq_b. Теперь заметим, что итерации цикла, считающие нужную мне статистику, независимы, а количество операций может быть довольно большим, если у картинки немалый размер, и можно сделать подсчет параллельным. Внутри каждого треда я также создам три массива размера 256 (thread_freq_r, thread_freq_g, thread_freq_b), посчитаю статистику для каждого треда, а потом создам

критическую секцию, внутри которой сложу все насчитанные значения в изначальные массивы freq_r, freq_g, freq_b. С помощью критической секции мне удается избежать состояния гонок.

Поскольку дальнейший код будет использовать данные массивов freq_r, freq_g, freq_b, то мы вынуждены поставить barrier, чтобы эти данные были корректны с вероятностью 100%.

Вспомним про коэффициент, который нам подали на входе. Нужно определить, какие цвета мы будем игнорировать. Здесь нет никаких хитростей – просто считаем префиксные/суффиксные суммы и проверяем на каждой итерации, не превысило ли отношение суммы к общему количеству пикселей наш коэффициент. Всего будет 6 циклов – левая и правая границы для каждого из трех цветов. Среди полученных левых границ возьмем минимальную – это будет переменная left_border, а среди правых – максимальную, это будет переменная right_border. Так нужно сделать по условию.

Однако, здесь все же есть тонкий момент. Допустим, что коэффициент равен 0.1 и на некотором префиксе находится 9% пикселей, а если мы увеличим количество пикселей на префиксе на 1, то на таком префиксе будет 11% пикселей. Какой из этих префиксов нужно игнорировать? В моей реализации я буду игнорировать меньший префикс по одной причине – чтобы избежать неприятной ситуации, когда отрезок может состоять из одного цвета.

После этого я предпосчитаю для каждого из 256 цветов – чему должно равняться значение этого цвета после изменения контрастности.

Эта информация будет храниться в массиве stretched. Формула того, как именно растягивается отрезок, была в теоретической части.

Итого, после первого распараллеленного цикла у меня есть еще семь. Но их я параллелить не стал. Причина очевидна – все эти циклы имеют по 256 итераций, что пренебрежимо мало и не повлияет на скорость алгоритма. После этого у меня опять будет параллельный цикл, поэтому все эти 7 циклов я обернул в single.

Последний штрих перед выводом – нужно просто сопоставить старому значению цвета новое. Этот цикл можно спокойно распараллелить, так как все итерации независимы. Перед циклом снова поставим barrier, чтобы все значения stretched были посчитаны корректно.

Теперь предстоит узнать время работы моей программы. Сразу оговорюсь, что у меня далеко не самый топовый ноутбук, и порой время двух последовательных запусков может отличаться в 2 раза. Тем не менее, я постарался взять усредненные значения времени, которые мне удалось получить, если делать перерыв 20 секунд между запусками программы. Тестировать буду на цветной картинке размером 11333 на 8500 пикселей и на 8-ядерном процессоре. При тестировании был использован флаг “O3” (который, кстати, ускоряет мою программу в 2-6 раз).

1) Посмотрим, как число потоков влияет на скорость выполнения. Все циклы в моей программе используют schedule(static).

Таблица №1 – Результаты тестов на разном количестве тредов.

Количество тредов	1 запуск, мс	2 запуск, мс	3 запуск, мс	Среднее значение, мс
1	1264	1250	1212	1242
2	658	671	653	661
3	454	454	434	447
4	363	371	354	362
6	220	226	247	231
8	147	184	149	160
16	166	179	172	172
32	161	167	152	160

Для наглядности приведу график:

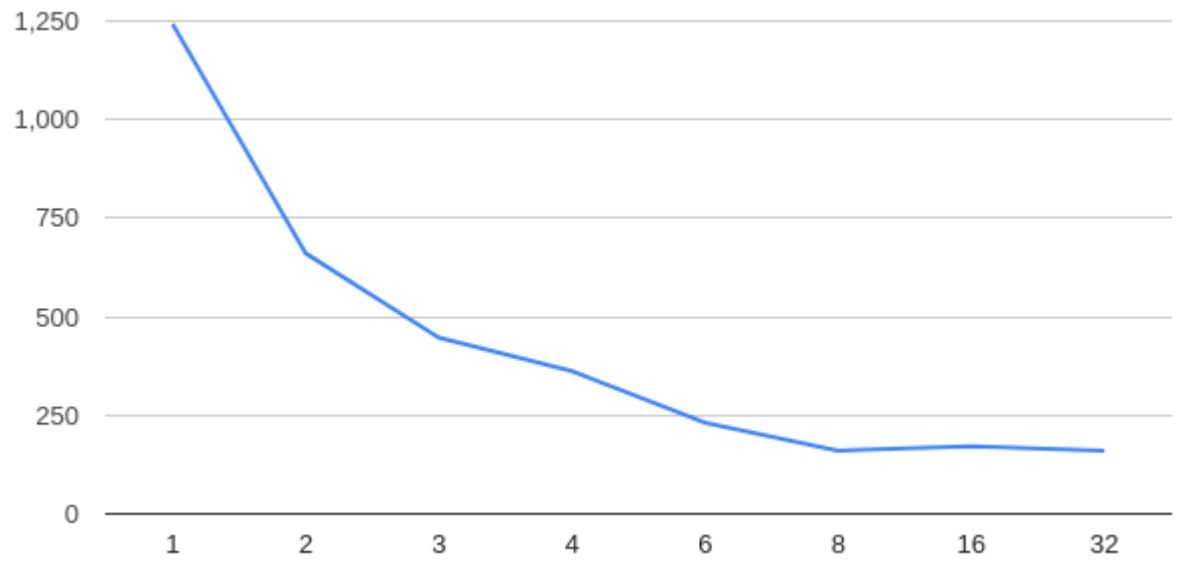

Рисунок №4 – Зависимость времени работы программы в миллисекундах (ось ординат) от количества тредов (ось абсцисс).

В целом, довольно ожидаемо. Поскольку мой процессор восьмиядерный, то прирост скорости был явно заметен до 8 тредов. После этого, на большем количестве тредов, время работы программы существенно не менялось.

2) Посмотрим, как программа будет работать при различных параметрах schedule. Количество тредов установим, например, на 32.

Таблица №2 – Результаты тестов с различными schedule.

schedule	1 запуск, мс	2 запуск, мс	3 запуск, мс	Среднее значение, мс
static(1)	1216	1241	1202	1220
dynamic(1)	7716	7721	7751	7729
static(32)	378	392	431	400
dynamic(32)	370	405	366	380
static(1024)	166	164	167	166
dynamic(1024)	169	147	149	155

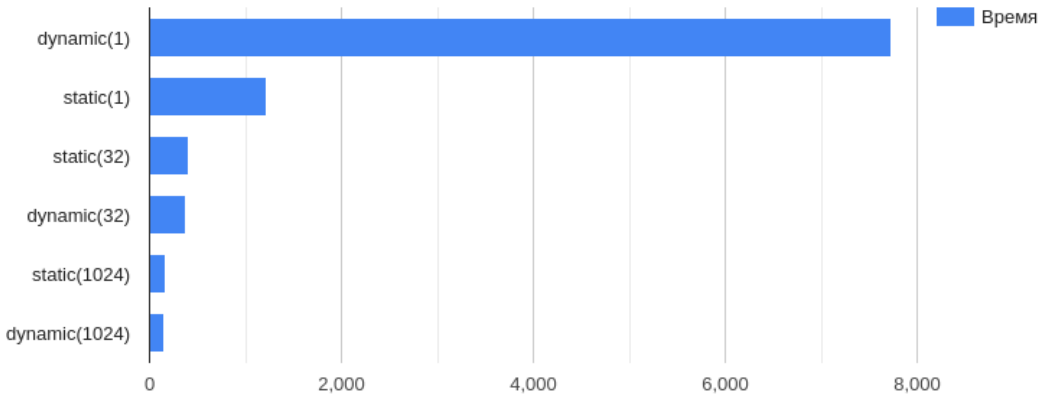

Рисунок №5 – Диаграмма полученных результатов с различными schedule.

Отсюда делаем вывод, что chunk_size стоит все-таки ставить побольше. И это неудивительно – количество кэш-попаданий резко возрастает, а время работы – наоборот. А dynamic имеет смысл только для больших chunk_size, причем он может быть даже лучше, чем static.

3) Попробуем запустить программу с включенным OpenMP и 1 тредом и с выключенным OpenMP. В процессе тестирования выяснилось, что если оставить флаг “O3”, то разницы практически не будет. Поэтому здесь я тестировал без этого флага (заодно можно увидеть, насколько сильно “O3” ускоряет мою программу).

Таблица №3 – Результаты тестов с включенным/выключенным OpenMP.

OpenMP	1 запуск, мс	2 запуск, мс	3 запуск, мс	Среднее значение, мс
вкл.	4432	4429	4416	4426
выкл.	4309	4316	4317	4314

Разница небольшая, но есть, что вполне логично. С включенным OpenMP время дополнительно тратится на создание треда.

Важное примечание: свой код я компилировал с флагами: -std=c++11 -fopenmp -O3. Если убрать флаг -fopenmp, то код тоже будет компилироваться и будет работать как в третьем тестировании, без OpenMP. Если убрать флаг -O3, то код будет дико тормозить (на 8 тредах я получал разницу по скорости в 6 раз), хотя тоже будет работать.

Листинг

Компилятор: g++-9.

main.cpp

```cpp
#include <omp.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>

using namespace std;

int main(int argc, char** argv) {
    // Reading and checking input
    if (argc != 5) {
        cout << "Program must be run with 4 arguments only" << endl;
        return 0;
    }
    int threads;
    try {
        threads = stoi(argv[1]);
    } catch (invalid_argument &e) {
        cout << "Number of threads must be an integer" << endl;
        return 0;
    } catch (out_of_range &e) {
        cout << "The number of threads is out of range" << endl;
        return 0;
    }
    if (threads < 0) {
        cout << "Number of threads mustn't be negative" << endl;
        return 0;
    }
    #ifdef _OPENMP
        if (threads != 0) {
            omp_set_num_threads(threads);
```

```
    }
#endif
string input_file_name = argv[2];
string output_file_name = argv[3];
double coeff;
try {
    coeff = stod(argv[4]);
} catch (invalid_argument &e) {
    cout << "Coefficient must be a real number" << endl;
    return 0;
} catch (out_of_range &e) {
    cout << "Coefficient is out of range" << endl;
    return 0;
}
if (coeff < 0 || coeff >= 0.5) {
    cout << "Coefficient must belong to semi-interval [0; 0.5)" << endl;
    return 0;
}
// Reading header of the input file and checking
ifstream input;
input.open(input_file_name, ios::binary);
if (!input.is_open()) {
    cout << "File cannot be opened :(" << endl;
    return 0;
}
string file_type;
int width, height, max_color;
try {
    input >> file_type >> width >> height >> max_color;
} catch (ios_base::failure &e) {
    cout << "File header cannot be read :(" << endl;
    input.close();
```

```
        return 0;
    }
    if (file_type != "P5" && file_type != "P6") {
        cout << "Unsupported file format" << endl;
        input.close();
        return 0;
    }
    if (width <= 0 || height <= 0) {
        cout << "Incorrect width/height of the image" << endl;
        input.close();
        return 0;
    }
    if (max_color != 255) {
        cout << "Incorrect max color value" << endl;
        input.close();
        return 0;
    }
    if (file_type == "P5") {
        vector<unsigned char> pic(height * width);
        try {
            input.get(); // The first symbol is always garbage
            for (int i = 0; i < height * width; i++) {
                pic[i] = input.get();
            }
        } catch (ios::failure &e) {
            cout << "Error while reading the image :(" << endl;
            input.close();
            return 0;
        }
        input.close();
        // Algorithm
        vector<int> freq(max_color + 1, 0);
```

```
        vector<unsigned char> stretched(max_color + 1);
        #ifdef _OPENMP
            double before_algo_time = omp_get_wtime();
                 #pragma omp parallel default(none) shared(height, width,
max_color, pic, freq, coeff, stretched)
            {
                vector<int> thread_freq(max_color + 1, 0);
                #pragma omp for schedule(static)
                for (int i = 0; i < height * width; i++) {
                    thread_freq[pic[i]]++;
                }
                #pragma omp critical
                {
                    for (int i = 0; i <= max_color; i++) {
                        freq[i] += thread_freq[i];
                    }
                }
                #pragma omp barrier
                #pragma omp single
                {
                    int psum = freq[0];
                    unsigned char left_border = max_color;
                    unsigned char right_border = 0;
                    for (int i = 1; i <= max_color; i++) {
                        psum += freq[i];
                        if (float(psum) / float(height * width) >= coeff) {
                            left_border = i - 1;
                            break;
                        }
                    }
                    psum = freq[max_color];
                    for (int i = max_color - 1; i >= 0; i--) {
```

```
                    psum += freq[i];
                    if (float(psum) / float(height * width) >= coeff) {
                        right_border = i + 1;
                        break;
                    }
                }
                for (int i = 0; i <= max_color; i++) {
                    if (i < left_border) {
                        stretched[i] = 0;
                    } else if (i > right_border) {
                        stretched[i] = max_color;
                    } else {
                                    float res = float(i - left_border) /
float(right_border - left_border) * max_color;
                        stretched[i] = (unsigned char)(res);
                    }
                }
            }
            #pragma omp barrier
            #pragma omp for schedule(static)
            for (int i = 0; i < height * width; i++) {
                pic[i] = stretched[pic[i]];
            }
        }
        double after_algo_time = omp_get_wtime();
        printf("Time (%i thread(s)): %g ms\n", threads, (after_algo_time
- before_algo_time) * 1000);
    #else
        auto start = std::chrono::system_clock::now();
        for (int i = 0; i < height * width; i++) {
            freq[pic[i]]++;
        }
```

```
            int psum = freq[0];
            unsigned char left_border = max_color;
            unsigned char right_border = 0;
            for (int i = 1; i <= max_color; i++) {
                psum += freq[i];
                if (float(psum) / float(height * width) >= coeff) {
                    left_border = i - 1;
                    break;
                }
            }
            psum = freq[max_color];
            for (int i = max_color - 1; i >= 0; i--) {
                psum += freq[i];
                if (float(psum) / float(height * width) >= coeff) {
                    right_border = i + 1;
                    break;
                }
            }
            for (int i = 0; i <= max_color; i++) {
                if (i < left_border) {
                    stretched[i] = 0;
                } else if (i > right_border) {
                    stretched[i] = max_color;
                } else {
                     float res = float(i - left_border) / float(right_border
- left_border) * max_color;
                    stretched[i] = (unsigned char)(res);
                }
            }
            for (int i = 0; i < height * width; i++) {
                pic[i] = stretched[pic[i]];
            }
```

```
        auto end = std::chrono::system_clock::now();
        std::chrono::duration<double> diff = end - start;
        printf("Time (no threads): %g ms\n", diff.count() * 1000);
    #endif
    // End of algorithm
    // Output
    ofstream output;
    output.open(output_file_name);
    output << "P5\n" << width << " " << height << "\n255\n";
    for (int i = 0; i < width * height; i++) {
        output.put(char(pic[i]));
    }
    output.close();
    return 0;
}
vector<unsigned char> pic(height * width * 3);
try {
    input.get(); // The first symbol is always garbage
    for (int i = 0; i < height * width * 3; i++) {
        pic[i] = input.get();
    }
} catch (ios::failure &e) {
    cout << "Error while reading the image :(" << endl;
    input.close();
    return 0;
}
input.close();
// Algorithm
vector<int> freq_r(max_color + 1, 0);
vector<int> freq_g(max_color + 1, 0);
vector<int> freq_b(max_color + 1, 0);
vector<unsigned char> stretched(max_color + 1);
```

```
    #ifdef _OPENMP
        double before_algo_time = omp_get_wtime();
         #pragma omp parallel default(none) shared(height, width, max_color,
pic, freq_r, freq_b, freq_g, coeff, stretched)
        {
            vector<int> thread_freq_r(max_color + 1, 0);
            vector<int> thread_freq_g(max_color + 1, 0);
            vector<int> thread_freq_b(max_color + 1, 0);
            #pragma omp for schedule(static)
            for (int i = 0; i < height * width * 3; i += 3) {
                thread_freq_r[pic[i]]++;
                thread_freq_g[pic[i + 1]]++;
                thread_freq_b[pic[i + 2]]++;
            }
            #pragma omp critical
            {
                for (int i = 0; i <= max_color; i++) {
                    freq_r[i] += thread_freq_r[i];
                    freq_g[i] += thread_freq_g[i];
                    freq_b[i] += thread_freq_b[i];
                }
            }
            #pragma omp barrier
            #pragma omp single
            {
                int psum = freq_r[0];
                unsigned char left_border_r = max_color;
                unsigned char right_border_r = 0;
                for (int i = 1; i <= max_color; i++) {
                    psum += freq_r[i];
                    if (float(psum) / float(height * width) >= coeff) {
                        left_border_r = i - 1;
```

```
            break;
        }
    }
    psum = freq_r[max_color];
    for (int i = max_color - 1; i >= 0; i--) {
        psum += freq_r[i];
        if (float(psum) / float(height * width) >= coeff) {
            right_border_r = i + 1;
            break;
        }
    }

    psum = freq_g[0];
    unsigned char left_border_g = max_color;
    unsigned char right_border_g = 0;
    for (int i = 1; i <= max_color; i++) {
        psum += freq_g[i];
        if (float(psum) / float(height * width) >= coeff) {
            left_border_g = i - 1;
            break;
        }
    }
    psum = freq_g[max_color];
    for (int i = max_color - 1; i >= 0; i--) {
        psum += freq_g[i];
        if (float(psum) / float(height * width) >= coeff) {
            right_border_g = i + 1;
            break;
        }
    }

    psum = freq_b[0];
```

```
                unsigned char left_border_b = max_color;
                unsigned char right_border_b = 0;
                for (int i = 1; i <= max_color; i++) {
                    psum += freq_b[i];
                    if (float(psum) / float(height * width) >= coeff) {
                        left_border_b = i - 1;
                        break;
                    }
                }
                psum = freq_b[max_color];
                for (int i = max_color - 1; i >= 0; i--) {
                    psum += freq_b[i];
                    if (float(psum) / float(height * width) >= coeff) {
                        right_border_b = i + 1;
                        break;
                    }
                }
                        unsigned char left_border = min(left_border_b,
min(left_border_g, left_border_r));
                       unsigned char right_border = max(right_border_b,
max(right_border_g, right_border_r));
                for (int i = 0; i <= max_color; i++) {
                    if (i < left_border) {
                        stretched[i] = 0;
                    } else if (i > right_border) {
                        stretched[i] = max_color;
                    } else {
                                    float res = float(i - left_border) /
float(right_border - left_border) * max_color;
                        stretched[i] = (unsigned char)(res);
                    }
                }
            }
```

```
            #pragma omp barrier
            #pragma omp for schedule(static)
            for (int i = 0; i < height * width * 3; i++) {
                pic[i] = stretched[pic[i]];
            }
        }
        double after_algo_time = omp_get_wtime();
         printf("Time (%i thread(s)): %g ms\n", threads, (after_algo_time -
before_algo_time) * 1000);
    #else
        auto start = std::chrono::system_clock::now();
        for (int i = 0; i < height * width * 3; i += 3) {
            freq_r[pic[i]]++;
            freq_g[pic[i + 1]]++;
            freq_b[pic[i + 2]]++;
        }
        int psum = freq_r[0];
        unsigned char left_border_r = max_color;
        unsigned char right_border_r = 0;
        for (int i = 1; i <= max_color; i++) {
            psum += freq_r[i];
            if (float(psum) / float(height * width) >= coeff) {
                left_border_r = i - 1;
                break;
            }
        }
        psum = freq_r[max_color];
        for (int i = max_color - 1; i >= 0; i--) {
            psum += freq_r[i];
            if (float(psum) / float(height * width) >= coeff) {
                right_border_r = i + 1;
                break;
```

```
    }
}

psum = freq_g[0];
unsigned char left_border_g = max_color;
unsigned char right_border_g = 0;
for (int i = 1; i <= max_color; i++) {
    psum += freq_g[i];
    if (float(psum) / float(height * width) >= coeff) {
        left_border_g = i - 1;
        break;
    }
}
psum = freq_g[max_color];
for (int i = max_color - 1; i >= 0; i--) {
    psum += freq_g[i];
    if (float(psum) / float(height * width) >= coeff) {
        right_border_g = i + 1;
        break;
    }
}

psum = freq_b[0];
unsigned char left_border_b = max_color;
unsigned char right_border_b = 0;
for (int i = 1; i <= max_color; i++) {
    psum += freq_b[i];
    if (float(psum) / float(height * width) >= coeff) {
        left_border_b = i - 1;
        break;
    }
}
```

```
        psum = freq_b[max_color];
        for (int i = max_color - 1; i >= 0; i--) {
            psum += freq_b[i];
            if (float(psum) / float(height * width) >= coeff) {
                right_border_b = i + 1;
                break;
            }
        }
          unsigned char left_border = min(left_border_b, min(left_border_g,
left_border_r));
        unsigned char right_border = max(right_border_b, max(right_border_g,
right_border_r));
        for (int i = 0; i <= max_color; i++) {
            if (i < left_border) {
                stretched[i] = 0;
            } else if (i > right_border) {
                stretched[i] = max_color;
            } else {
                  float res = float(i - left_border) / float(right_border -
left_border) * max_color;
                stretched[i] = (unsigned char)(res);
            }
        }
        for (int i = 0; i < height * width * 3; i++) {
            pic[i] = stretched[pic[i]];
        }
        auto end = std::chrono::system_clock::now();
        std::chrono::duration<double> diff = end - start;
        printf("Time (no threads): %g ms\n", diff.count() * 1000);
    #endif
    // End of algorithm
    // Output
    ofstream output;
```

```
    output.open(output_file_name, ios::binary);
    output << "P6\n" << width << " " << height << "\n255\n";
    for (int i = 0; i < width * height * 3; i++) {
        output.put(char(pic[i]));
    }
    output.close();
    return 0;
}
```