САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

«OpenMp»

Выполнил: Рассадников Григорий

Номер ИСУ: 334838

студ. гр. M3135(M3138)

Санкт-Петербург

2021–2022

Цель работы: знакомство со стандартом OpenMP.

Теоретическая часть

1. **Основные сведения**

OpenMP – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций. Позволяет достаточно легко создавать многопоточные приложения на C/C++, Fortran. Основной поток порождает дочерние потоки по мере необходимости.

Программирование путем вставки директив компилятора в ключевые места исходного кода программы. Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для распараллеливания участков кода. Рисунок 1 – пример распараллеливания.

Последовательный код

```
void main(){
  double x[1000];
  for(i=0; i<1000; i++){
    calc_emth(&x[i]);
  }
}
```

Параллельный код

```
void main(){
  double x[1000];
#pragma omp parallel for ...
  for(i=0; i<1000; i++){
    calc_smth(&x[i]);
  }
}
```

Рисунок 1 – пример распараллеливания

Директива **#pragma omp parallel for** указывает на то, что данный цикл следует разделить по итерациям между потоками. Количество потоков можно контролировать из программы, или через среду выполнения программы13 – переменную окружения **OMP_NUM_THREADS**.

Следует отметить, что разработчик ответственен за синхронизацию потоков и зависимость между данными. Для того, чтобы скомпилировать программу с поддержкой OpenMP компилятору следует указать дополнительный ключ: **-fopenmp** (в нашем случае, работа пишется на C++).

В C/C++ следует подключить файл заголовков omp.h: **#include<omp.h>**

2. Синтаксис

В основном конструкции OpenMP – это директивы компилятора. Для C/C++ директивы имеют следующий вид: **#pragma omp конструкция [условие [условие]...]**

3. Условия выполнения в конструкциях OpenMp

Условия выполнения определяют то, как будет выполняться параллельный участок кода и область видимости переменных внутри этого участка кода.

- **shared(var1, var2, ...)**

  Условие **shared** указывает на то, что все перечисленные переменные будут разделяться между потоками. Все потоки будут доступаться к одной и той же области памяти.
- **private(var1, var2, ...)**

  Условие **private** указывает на то, что каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.
- **firstprivate(var1, var2, ...)**

  Это условие аналогично условию **private** за тем исключением, что указанные переменные инициализируются при входе в параллельный участок кода значением, которое имела переменна до входа в параллельную секцию.
- **lastprivate(var1, var2, ...)**

  Приватные переменные сохраняют свое значение, которое они получили при достижении конца параллельного участка кода.
- **reduction(оператор:var1, var2, ...)**

  Это условие гарантирует безопасное выполнение операций редукции, например вычисление глобальной суммы. Приватная копия каждой перечисленной переменной инициализируется при входе в

параллельную секцию в соответствии с указанным оператором. При выходе из параллельной секции из частично вычисленных значений вычисляется результирующее и передается в основной поток. Доступны следующие операторы и агрегатные функции в C/C++: **+, -, *, &, ^, |, &&, || min, max**. Рисунок 2 – пример использования reduction.

```
#pragma omp parallel
{
#pragma for shared(x) private(i) reduction(+:sum)
    for(i=0; i<10000; i++)
          sum += x[i];
}
```

Рисунок 2 – пример использования reduction

- **schedule(тип [, размер блока])**

  Этим условием контролируется то, как итерации цикла распределяются между потоками. Тип расписания может принимать следующие значение (мы рассмотрим только 2, хотя существует несколько других):

  - ✓ **static** – итерации равномерно распределяются по потокам. Т.е. если в цикле 1000 итераций и 4 потока, то один поток обрабатывает все итерации с 1 по 250, второй – с 251 по 500, третий - с 501 по 750, четвертый с 751 по 1000. Если при этом задан еще и размер блока, то все итерации блоками заданного размера циклически распределяются между потоками. Статическое распределение работы эффективно, когда время выполнения итераций равно, или приблизительно равно.
  - ✓ **dynamic** – работа распределяется пакетами заданного размера между потоками. Как только какой-либо из потоков заканчивает обработку своей порции данных, он захватывает следующую. Стоит отметить, что при этом подходе несколько большие накладные расходы, но

можно добиться лучшей балансировки загрузки между потоками.

4. Конструкция for в OpenMP для распределения работ

Цель конструкции – распределение итераций цикла по потокам. Директива параллельного цикла **for** имеет следующий синтаксис для C/C++:

#pragma omp for [условие [,условие] ...]
цикл for ….

Рисунок 3 – пример конструкции for.

```
#pragma omp parallel
{
#pragma omp for private(i) shared(a,b)
  for(i=0; i<10000; i++)
    a[i] = a[i] + b[i]
}
```

Рисунок 3 – пример конструкции for

Еще можно распараллелить циклы (или другие конструкции) между собой, для этого используется секции со следующем синтаксисом:

#pragma omp parallel sections
{
#pragma omp section
{ //код для первой секции
}
#pragma omp section
{ //код для второй секции
}
…
}

5. Переменные окружения OpenMP

- **OMP_NUM_THREADS**

Устанавливает количество потоков в параллельном блоке. По умолчанию, количество потоков равно количеству виртуальных процессоров.

- **OMP_SCHEDULE**

  Устанавливает тип распределения работ в параллельных циклах с типом runtime.

- **OMP_DYNAMIC**

  Разрешает или запрещает динамическое изменение количества потоков, которые реально используются для вычислений (в зависимости от загрузки системы). Значение по умолчанию зависит от реализации.

- **OMP_NESTED**

  Разрешает или запрещает вложенный параллелизм (распараллеливание вложенных циклов). По умолчанию – запрещено.

По сути, в своей работе я использую только **OMP_NUM_THREADS**. Но остальные переменные окружения полезны при дальнейшем изучении OpenMp.

6. **Некоторые функции OpenMP**

   Здесь собраны некоторые функции, которые используются в коде программы, или использовались мной при отладке или тестировании.

- **void omp_set_num_threads(int num_threads)**

  Устанавливает количество потоков, которое может быть запрошено для параллельного блока.

- **int omp_get_num_threads()**

  Возвращает количество потоков в текущей команде параллельных потоков.

- **int omp_get_max_threads()**

  Возвращает максимальное количество потоков, которое может быть установлено **omp_set_num_threads**.

- **int omp_get_thread_num()**

  Возвращает номер потока в команде (целое число от 0 до количества потоков – 1).

- **double omp_get_wtime()**

  Возвращает значение, равное времени, прошедшему настенным часам в секундах с некоторого времени в прошлом.

Практическая часть (Вариант – Hard)

Используемый язык: C++. ОС: linux.

Запуск кода: g++ -fopenmp hw5.cpp -o hw5 && ./hw5 <число потоков> <имя_входного файла> <имя_выходного_файла> <коэффициент>

Сначала обработаем ошибки ввода, корректность файла (согласно документации), достанем ширину и высоту изображения. Считаем данные о цветах пикселей в массив **picture**. Заведем необходимые переменные. обработка входного файла. Рисунок 4 – обработка данных.

```
ifstream input;
ofstream output;
vector<int> picture;
double k;
int num_threads;
int width;
int height;
string p_6;
string check_255;

void open_files(int argc, char *argv[]) {...}
void read_in() {...}
void check_ppm(string p_6, string check_255) {...}
void read_picture() {...}
```

Рисунок 4 – обработка данных

Напишем функции **make_new_gray_picture()** и **make_new_color_picture()**, которые будут изменять контрастность нашего черно-белого или цветного изображения. В случае обработки черно-белого изображения алгоритм

очевидно вытекает из цветного, т. е. мы имеем всего один канал, что легко реализуется, зная алгоритм для трех.

Основная идея алгоритма: Давайте возьмем один пиксель и рассмотрим яркость его каждого пикселя. Т.к. яркость принимает значение от 0...255, то заведем массив counter_colors_r, counter_colors_g, counter_colors_r, которые будут хранить в i-ой ячейки, сколько всего пикселей имеют яркость **i**(для каждого канала отдельно). Теперь согласно коэффициенту **k**, введённым пользователем, пропустим сначала и с конца массива **counter_colors_r/g/ b** часть пикселей равное **k*size_picture/3**. Т. е. пропустим часть самых яркий и самых темный пикселей. Теперь узнаем яркость самого темного и самого светлого пикселя в каждом канале, не учитывая пропущенные: **color_start_r/g/b**, **color_end_r/g/b**. Чтобы корректно обработать изображения и не исказить картинку, выберем границу начального и конечного цвета следующим образом: **color_start = min(color_start_r, color_start_g, color_start_b) color_end = max(color_end_r, color_end_g, color_end_b)**. Теперь растянем пиксель из диапазона **color_start…color_end** до диапазона 0…255. Используя простую зависимость для пикселя **i**:

$$\frac{\mathbf{255}}{\boldsymbol{color\_end\ -\ color\_start}} = \frac{\boldsymbol{color}}{\boldsymbol{i - color\_start}}$$

где color – новый цвет. Т. е. мы можем получить массив **convert**, в котором будет значение каким должен стать пиксель, имеющий яркость **i** за 256 итераций цикла (конечно дробные значения становятся целыми, значения большие 255 становятся равны 255, а меньшие 0 равны 0). После этого пробежимся по всем значениям picture, и присвоим им новые значения **(picture[i] = convert[picture[i]])**.

Замерим время перед запуском функции **make_new_picture()** и после ее исполнения используя метод **omp_get_wtime()**, который возвращает время в секундах (это и будет время затраченное алгоритмом). В случае если

программа запускалась без ключа **-fopenmp**, то замеряем время стандартным **clock()**. Затем выведем данные в выходной файл, и закроем файлы. Рисунок 5 – замер времени работы с ключом.

```
double start_t = omp_get_wtime();

make_new_picture();

double end_t = omp_get_wtime();
double t = end_t - start_t;
printf( Format: "Time (%i thread(s)): %g ms\n", num_threads, (double) (t * 1000));

write();
input.close();
output.close();
```

Рисунок 5 – замер времени работы с ключом

Рисунок 6 – пример работы программы, при различных значениях коэффициента (исходник, k=0; 0.1; 0.2; 0.3; 0.4; 0.49999).

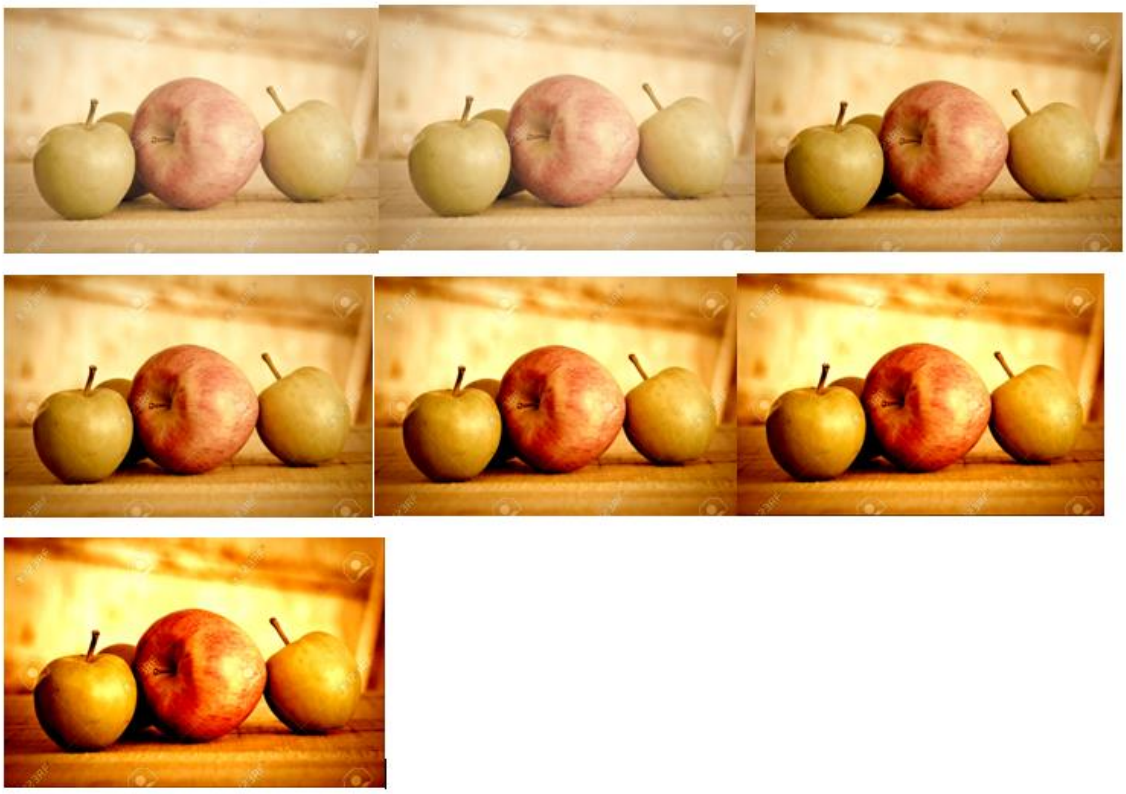

Рисунок 6 – пример работы программы

Теперь используя OpenMp, установим число потоков (используя функцию **omp_set_num_threads**, если число потоков больше 0), распараллелим

некоторые циклы **for** в функции **make_new_color/gray_picrute**. Также будем использовать секции, для разделения циклов по потокам.

- Заполнение массива counter_colors (Рисунок 7).

  Делаем копию переменной i для каждого потока, и глобально counter_colors, picture. Используя секции, разбиваем циклы на разные потоки.

```
#pragma omp parallel sections
    {

#pragma omp section
        {
#pragma omp parallel for shared(counter_colors_r, picture) schedule(guided)
            for (i = 0; i < picture_size; i = i + 3) {
                counter_colors_r[picture[i]]++;
            }
        }

#pragma omp section
        {
#pragma omp parallel for shared(counter_colors_g, picture) schedule(guided)
```

Рисунок 7 – Заполнение массива counter_colors_r/g/b

- Заполнение массива convert (Рисунок 8).

```
#pragma omp parallel for shared(convert, color_end, color_start) private(color, i)
    for (i = 0; i < 256; i++) {
        color = (i - color_start) * 255 / (color_end - color_start);
        if (color > 255) {
            color = 255;
        }
        if (color < 0) {
            color = 0;
        }
        convert[i] = color;
    }
```

Рисунок 8 – Заполнение массива convert

- Заполнение picture новыми значениями (Рисунок 9).

```
#pragma omp parallel for shared(convert, picture) private(i,cop)
    for ( i = 0; i < picture_size; i++) {
        cop = picture[i];
        picture[i] = convert[cop];
    }
```

Рисунок 9 – Заполнение picture новыми значениями

В случае с черно-белой, все очень похоже. В этом случае не нужно использовать секции т. к. канал всего один, что упрощает работу.

Используя условие выполнения schedule() в каждом параллельном цикле, протестируем программу на типе dynamic, static, различных значениях разделения блока при параллельности. Протестируем программу при использовании разного числа потоков (или использовании одного главного потока).

Рисунок 10 – зависимость работы от числа потоков.

Рисунок 11 – зависимость работы от коэффициента schedule равного части глубины каждого цикла.

Рисунок 12– зависимость работы от коэффициента schedule равного числу.

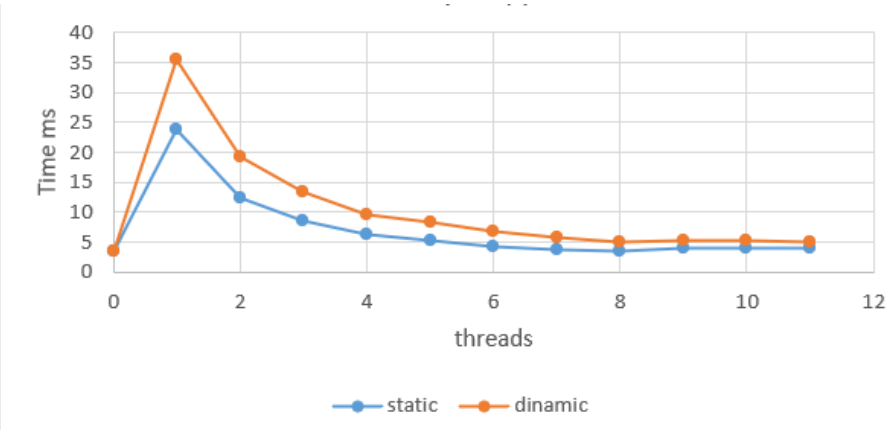

Рисунок 10 – зависимость работы от числа потоков

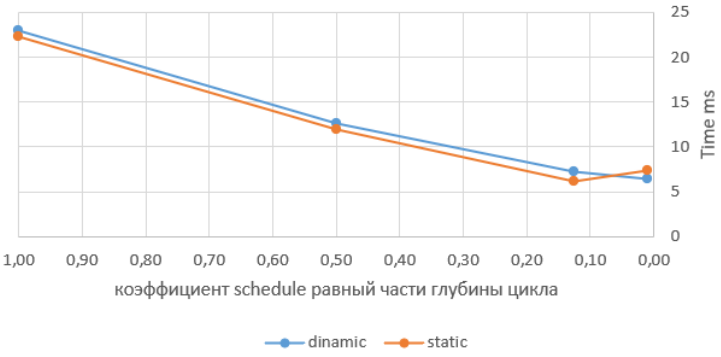

Рисунок 11 – зависимость работы от коэффициента schedule равного части глубины каждого цикла

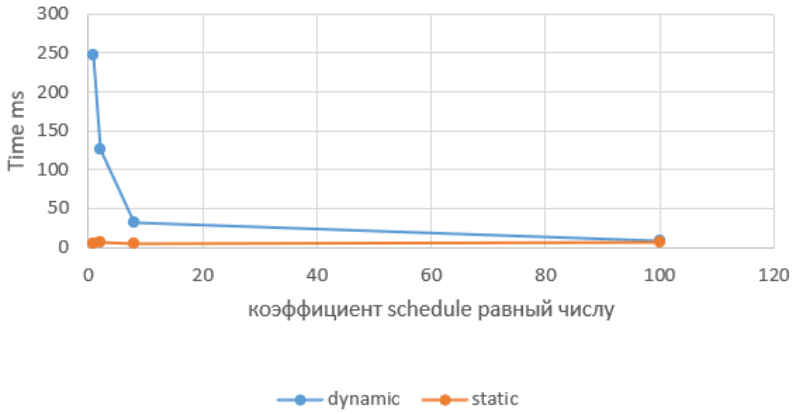

Рисунок 12 – зависимость работы от коэффициента schedule равного числу

Листинг кода

hw5.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <ctime>
#include <omp.h>



using namespace std;
#define uchar unsigned char

ifstream input;
ofstream output;
vector<int> picture;
double k;
int num_threads;
int width;
int height;
string p_5_6;
string check_255;
int counter_colors_r[256];
int counter_colors_g[256];
int counter_colors_b[256];
```

```
void open_files(int argc, char *argv[]) {
    if (argc < 5) {
        cout << "please write count threads, input and output file name,
parameter\n";
        exit(0);
    }

    input.open(argv[2]);
    output.open(argv[3]);
    if (!input) {
        cout << "failed to find input file\n";
        exit(0);
    }

    if (!output) {
        cout << "failed to find output file\n";
        exit(0);
    }

    string _num_threads = argv[1];
    for (auto i:_num_threads) {
        if (i<'0' || i>'9') {
            cout << "number of threads must be int\n";
            exit(0);
        }
    }
    num_threads = stoi(_num_threads);
    string _k = argv[4];
    for (auto i:_k) {
        if ((i<'0' || i>'9')&& i!='.') {
            cout << "coefficient must be number(int,float,double...)\n";
            exit(0);
        }
    }
    k = stod(_k);
```

```
    if (k>=0.5|| k<0){
        cout << "coefficient must be in [0,0.5)\n";
        exit(0);
    }

}
void read_in() {
    char e;
    p_5_6 = "";
    input.get(e);
    while (e != '\n') {
        p_5_6 = p_5_6 + e;
        input.get(e);
    }
    input.get(e);
    string _width = "";
    string _height = "";
    while (e != ' ' && e != '\n') {
        _width = _width + to_string((uchar) e - 48);
        input.get(e);
    }
    input.get(e);
    while (e != '\n') {
        _height = _height + to_string((uchar) e - 48);
        input.get(e);
    }
    input.get(e);
    check_255 = "";
    while (e != '\n') {
        check_255 = check_255 + e;
        input.get(e);

    }

    width = atoi(_width.c_str());
```

```
    height = atoi(_height.c_str());

}
void check_ppm(string p_5_6, string check_255) {
    if (p_5_6 != "P6" && p_5_6 != "P5") {
        cout << "expected P6 in file header";
        exit(0);
    }

    if (check_255 != "255") {
        cout << "expected 255 in file header after size";
    }
}
void read_picture() {
    char e;
    while (input.get(e)) {
        picture.push_back((uchar) e);
    }
}




void make_new_gray_picture(){
    int picture_size = picture.size();
    int counter_colors[256];
    int propusk = k * picture_size;
    int _propusk = propusk;
    int color_start = 0;
    int color_end = 255;

#if defined(_OPENMP)
    if (num_threads > 0) {
        omp_set_num_threads(num_threads);
    }
#endif
```

```
    int i;

#pragma omp parallel for shared(counter_colors) schedule(guided)
    for (i = 0; i < 256; i++) {
        counter_colors[i] = 0;
    }

#pragma omp parallel for shared(counter_colors, picture) schedule(guided)
    for (i = 0; i < picture_size; i++) {
        counter_colors[picture[i]]++;
    }



    for ( i = 0; i < 256; i++) {
        _propusk -= counter_colors[i];
        if (_propusk < 0) {
            color_start = i;
            break;
        }
    }



    _propusk = propusk;

    for (i = 255; i >= 0; i--) {
        _propusk -= counter_colors[i];
        if (_propusk < 0) {
            color_end = i;
            break;
        }
    }

    if (color_start > color_end) {
        color_start = (color_start + color_end) / 2;
        color_end = color_start;
```

```
    }
    int convert[256];
    int color;
    double deletel = 255.0/(color_end - color_start+1);

#pragma omp parallel for shared(convert, color_end, color_start) schedule(guided)
    for (i = 0; i < 256; i++) {
        color = (i - color_start) * deletel;
        if (color > 255) {
            color = 255;
        }
        if (color < 0) {
            color = 0;
        }
        convert[i] = color;
    }
    int cop;

#pragma omp parallel for shared(convert, picture) schedule(guided)
    for ( i = 0; i < picture_size; i++) {
        cop = picture[i];
        picture[i] = convert[cop];
    }



}
void make_new_color_picture() {
    int picture_size = picture.size();
    int propusk = k * picture_size / 3;
    int _propusk_r = propusk;
    int _propusk_g = propusk;
    int _propusk_b = propusk;
    int color_start = 0;
    int color_end = 255;
    int color_start_r = 0;
```

```
    int color_end_r = 255;
    int color_start_g = 0;
    int color_end_g = 255;
    int color_start_b = 0;
    int color_end_b = 255;

#if defined(_OPENMP)
    if (num_threads > 0) {
        omp_set_num_threads(num_threads);
    }
#endif

#pragma omp parallel sections
    {

#pragma omp section
        {
#pragma omp parallel for shared(counter_colors_r, picture) schedule(guided)
            for (int i = 0; i < picture_size; i = i + 3) {
                counter_colors_r[picture[i]]++;
            }
        }

#pragma omp section
        {
#pragma omp parallel for shared(counter_colors_g, picture) schedule(guided)
            for (int i = 1; i < picture_size; i = i + 3) {
                counter_colors_g[picture[i]]++;
            }
        }
#pragma omp section
        {
#pragma omp parallel for shared(counter_colors_b, picture) schedule(guided)
            for (int i = 2; i < picture_size; i = i + 3) {
                counter_colors_b[picture[i]]++;
```

```
            }
        }
    }

#pragma omp parallel sections
    {
#pragma omp section
        {
            for (int i = 0; i < 256; i++) {
                _propusk_r -= counter_colors_r[i];
                if (_propusk_r < 0) {
                    color_start_r = i;
                    break;
                }
            }
        }
#pragma omp section
        {
            for (int i = 0; i < 256; i++) {
                _propusk_g -= counter_colors_g[i];
                if (_propusk_g < 0) {
                    color_start_g = i;
                    break;
                }
            }
        }
#pragma omp section
        {

            for (int i = 0; i < 256; i++) {
                _propusk_b -= counter_colors_b[i];
                if (_propusk_b < 0) {
                    color_start_b = i;
                    break;
                }
```

```
            }
        }
    }



    _propusk_r = propusk;
    _propusk_g = propusk;
    _propusk_b = propusk;

#pragma omp parallel sections
    {
#pragma omp section
        {
            for (int i = 255; i >= 0; i--) {
                _propusk_r -= counter_colors_r[i];
                if (_propusk_r < 0) {
                    color_end_r = i;
                    break;
                }
            }
        }
#pragma omp section
        {
            for (int i = 255; i >= 0; i--) {
                _propusk_g -= counter_colors_g[i];
                if (_propusk_g < 0) {
                    color_end_g = i;
                    break;
                }
            }
        }
#pragma omp section
        {
            for (int i = 255; i >= 0; i--) {
```

```
                _propusk_b -= counter_colors_b[i];
                if (_propusk_b < 0) {
                    color_end_b = i;
                    break;
                }

            }
        }
    }

    color_start = min(color_start_r,min(color_start_g,color_start_b));
    color_end = max(color_end_r,max(color_end_g,color_end_b));

    if (color_start > color_end) {
        color_start = (color_start + color_end) / 2;
        color_end = color_start;
    }
    int convert[256];
    int color;
    double deletel = 255.0/(color_end - color_start+1);

#pragma omp parallel for shared(convert, color_end, color_start) private(color)
schedule(guided)
    for (int i = 0; i < 256; i++) {
        color = (i - color_start)*deletel;
        if (color > 255) {
            color = 255;
        }
        if (color < 0) {
            color = 0;
        }
        convert[i] = color;
    }
    int cop;

#pragma omp parallel for shared(convert, picture) private(cop) schedule(guided)
```

```
    for (int i = 0; i < picture_size; i++) {
        cop = picture[i];
        picture[i] = convert[cop];
    }


}

void write() {
    output << p_5_6 << '\n';
    output << width << " " << height << '\n';
    output << 255 << '\n';

    for (auto i: picture) {
        output << (uchar) i;
    }
}

int main(int argc, char *argv[]) {
    open_files(argc, argv);
    read_in();
    check_ppm(p_5_6, check_255);
    read_picture();

#if defined(_OPENMP)
    double start_t = omp_get_wtime();
#else
    time_t start_t = clock();
#endif

    if (p_5_6=="P6") {
        make_new_color_picture();
    }else if (p_5_6=="P5"){
        make_new_gray_picture();
    }
```

```
#if defined(_OPENMP)
    double end_t = omp_get_wtime();
     double t = end_t - start_t;
    printf("Time (%i thread(s)): %g ms\n", num_threads, (double) (t * 1000));
#else
    time_t end_t = clock();
    time_t t = end_t - start_t;
    printf("Time without openmp: %g ms\n", (double) (t * 1000)/CLOCKS_PER_SEC);
#endif

    write();
    input.close();
    output.close();
    return 0;

}
```