САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

«OpenMP»

Выполнил: Ивченков Дмитрий Артемович

Номер ИСУ: 334906

студ. гр. М3134

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

Стандарт OpenMP

OpenMP – это стандарт и библиотека для параллельного программирования вычислительных систем с общей памятью. Официально поддерживаются C, C++ и Fortran, также существуют реализации для некоторых других языков (Java, Python, Pascal и т.д.). Библиотека предоставляет достаточно простой и гибкий интерфейс, позволяющий постепенно и удобно распараллеливать написанную изначально последовательную программу. В стандарте используется параллелизм потоков. Потоки (треды) создаются в рамках одного процесса и имеют собственную память и доступ к общей памяти процесса. Возможность распараллеливания применима только к независимым относительно друг друга действиям.

Программирование осуществляется путём вставки директив в места исходного кода, которые нужно распараллелить. При запуске программы создаётся единственный основной поток (master), работающий всё время работы процесса. Встретив параллельную область, задаваемую директивой #pragma omp parallel, он порождает ряд потоков. Далее каждый поток выполняет свои собственные и независимые действия. Границы параллельной области в коде определяются фигурными скобками. При выходе из области созданные потоки уничтожаются.

При объявлении параллельной области есть возможность установить ей некоторые характеристики. Определить необходимое количество создаваемых потоков можно, задав параметр num_threads положительным целочисленным значением. Количество потоков также задаётся вызовом функции библиотеки omp_set_num_threads. Кроме того, если перед выполнением определено значение

переменной окружения (environment variable) OMP_NUM_THREADS, то оно будет использоваться в качестве количества потоков при отсутствии других пунктов. В остальных случаях потоки создаются в количестве по умолчанию. Параллельной области можно задать ещё другие параметры: условие создания команды потоков, разрешение на создание вложенных потоков, доступ к переменным и прочие.

Созданные до объявления параллельной области переменные становятся общими для всех потоков, а переменные, созданные внутри потока, являются локальными (приватными) и доступны только текущему потоку. При одновременных попытках доступа к общим ресурсам возникает состояние гонки (race condition). В таком случае невозможно гарантировать корректность действий и результата, поэтому необходимо не допускать возникновения данной проблемы. Для её решения библиотека OpenMP предоставляет возможности объявления критических секций, атомарных операций и барьеров с помощью директив #pragma omp critical, atomic и barrier соответственно.

Для распараллеливания цикла в стандарте существует директива #pragma omp for. Параллельный цикл позволяет разбить его итерации по потокам, чтобы каждый выполнил свою часть. Для возможности распараллеливания цикл должен иметь каноническую форму и соответствовать определённым критериям, описанным в документации OpenMP. Принцип распределения итераций между потоками позволяет задать опция schedule, имеющая два параметра kind – тип планирования и chunk_size – размер порции. Опция schedule(static, chunk_size) определяет статическое планирование. Итерации цикла распределяются порциями размером chunk_size между потоками, пока не закончатся. Планирование выполняется один раз в начале, и каждый поток «узнаёт», какие итерации выполнит. Если chunk_size не указан, то итерации начиная с начала цикла примерно поровну будут распределены между потоками. Динамическое планирование задаётся при помощи schedule(dynamic, chunk_size). При динамическом планировании каждый поток получает заданное число итераций, выполняет их и запрашивает новую порцию. В отличие от статического

планирования, динамическое выполняется многократно во время выполнения программы. Распределение зависит от темпа работы потоков и трудоёмкости операций. По умолчанию поток берёт по одной итерации. Кроме того, существуют ещё опции планирования guided – с изменяемым размером порций итераций – и runtime, при которой решение о планировании откладывается до выполнения и определяется значением переменной окружения OMP_SCHEDULE. Также для параллельного цикла могут быть установлены другие параметры, такие как nowait, убирающий барьерную синхронизацию в конце цикла, и reduction, задающий операцию, которая будет выполнена над локальными переменными при выходе из параллельной области. Для задания параллельной области, содержащей только параллельный for, существует директива #pragma omp parallel for.

Автоматическая контрастность изображения

Если точки изображения принимают не все значения яркости в диапазоне от 0 до 255, т.е. диапазон значений яркости не максимален, то изображение имеет плохую контрастность. Значения пикселей можно изменить так, чтобы линейно растянуть диапазон значений яркости. Тогда изображение станет более контрастным. Чтобы растянуть диапазон, нужно найти минимальное y_{min} и максимальное y_{max} значения яркости. Далее из значения каждого пикселя вычесть минимальное значение, чтобы пиксели с наименьшими значениями достигли минимальной яркости и диапазон начался с нуля, а потом умножить каждое значение на коэффициент растяжения, равный отношению максимального требуемого значения яркости 255 к разности максимального и минимального значения в изображении. Таким образом, пиксели с наибольшими значениями достигнут максимальной яркости, и весь диапазон значений яркости линейно растянется. Алгоритм автоматического увеличения контрастности изображения заключается в применении к каждому пикселю формулы 1.

$$f(y) = (y - y_{\min}) \cdot \frac{255}{y_{max} - y_{min}} \quad (1)$$

Для цветных изображений в модели RGB необходимо найти минимальные и максимальные значения в каждом канале отдельно и для общего минимального и максимального значений взять минимум и максимум из этих трёх значений соответственно. Таким образом, каждый цвет изменится одинаково.

При вычислении растяжения стоит игнорировать некоторую долю по количеству самых тёмных и самых светлых точек изображения, которые можно считать шумом, чтобы они не помешали изменению контрастности, поэтому алгоритму нужно указывать коэффициент, чтобы не учитывать указанную долю точек. Также важно растяжение диапазона выполнять с насыщением, чтобы значения не вышли за границы от 0 до 255 и точки изображения не изменили своего смысла.

Практическая часть

Программа, выполняющая алгоритм автоматического увеличения контрастности, написана на языке C++ 14. Программе на вход подаются значения количества потоков, имя входного файла изображения, имя выходного файла изображения и значение коэффициента в указанном порядке. Для чтения из файла создаётся и открывается в бинарном режиме поток ifstream, для записи – ofstream. В первую очередь, считываются два идентификационных байта, позволяющих определить формат файла. P5 означает изображение в оттенках серого, а P6 – цветное изображение. Все действия над данными в программе производятся в зависимости от формата файла. Присутствует обработка исключений: если потоки не открыты и файлы не удаётся прочитать или если первые два байта не соответствуют требуемым форматам файла, то генерируется исключение с сообщением об ошибке. Далее, согласно структуре PNM файла, считываются ширина width и высота height изображения, а также максимальное значение пикселя maxval, в данном задании всегда равное 255.

После этого в массив считываются значения всех пикселей. Для изображения в оттенках серого значения яркости записываются в массив

gray_image. Для пикселей цветного изображения создана структура pixel, содержащая информацию о пикселе, и при чтении значения соответствующих цветовых компонент RGB пикселей записываются в массив image.

После чтения начинается замер времени работы программы. Момент времени определяется с помощью функции omp_get_wtime(). При помощи алгоритма нахождения k-ой порядковой статистики в массиве значений яркости изображения находится минимальное и максимальное значения с учётом коэффициента. Далее следует директива #pragma omp parallel, создающая параллельную область. Количество создаваемых потоков задаётся опцией num_threads с числом, поданным на вход программе. Если число не является положительным, в том числе, если число равно нулю, то количество потоков определяется по умолчанию. Внутри параллельной области указанием директивы #pragma omp for распараллеливается цикл. Задана опция schedule с параметром runtime. Планирование цикла определяется значением переменной окружения OMP_SCHEDULE, установленным в системе. В параллельном цикле выполняется вычисление новых значений яркости по формуле. Вычисления производятся с насыщением. После этого засекается время окончания работы.

В конце программы значения пикселей записываются в выходной файл, и потоки закрываются. Выводится время работы программы без учёта времени на чтение и запись.

Экспериментальная часть

Программа протестирована на различных изображениях. Ниже приведены примеры автоматического увеличения контрастности с коэффициентом изображения в оттенках серого (рисунок 1) и цветного изображения (рисунок 2). Слева показано исходное изображение, а справа – изображение с увеличенной контрастностью.

Рисунок № 1 – Результат алгоритма на изображении в оттенках серого

Рисунок № 2 – Результат алгоритма на цветном изображении

Проведены замеры времени работы программы при различных параметрах распараллеливания. Программа компилировалась с помощью компилятора GCC 11.2.0 командой g++ -O3 main.cpp -o hw5 -fopenmp. Замер проводился при работе программы с цветным изображением 7680×4320 пикселей и коэффициентом 0.1. На основе полученных данных построены графики для наглядного изображения зависимости времени работы программы от количества потоков и параметров планирования. Максимальное количество создаваемых потоков на компьютере выполняющего равно 8. На всех графиках для сравнения красной линией показано время работы программы с выключенным OpenMP. Графики на рисунках 3 и 4 показывают время работы программы при различных значениях числа потоков при одинаковом параметре schedule (static и dynamic соответственно).

Рисунок № 3 – График времени работы при различных значениях числа потоков при одинаковом schedule(static, chunk_size)

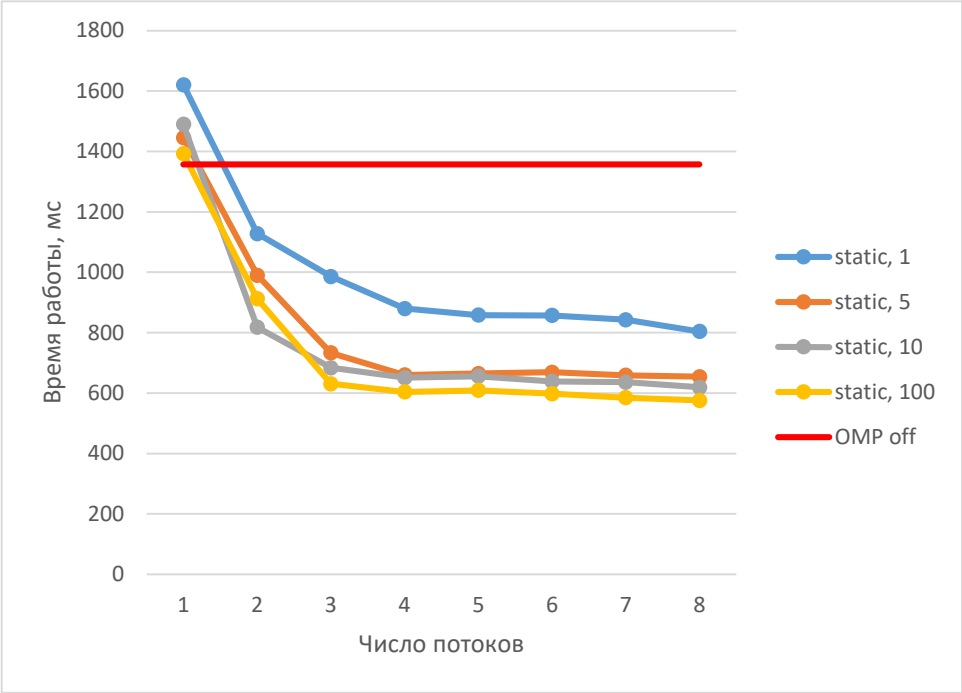
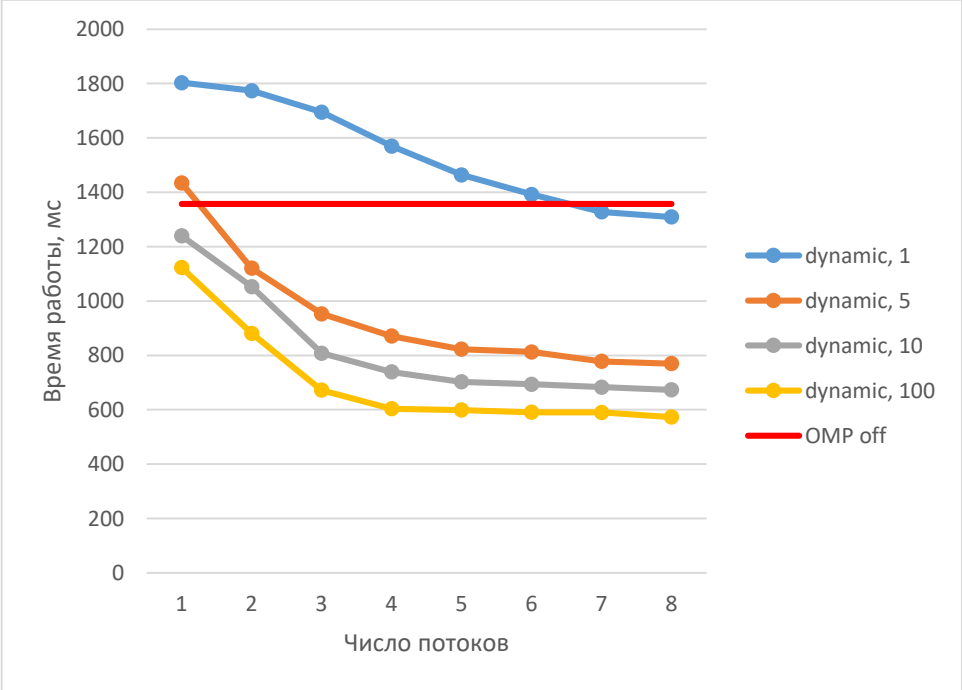


Рисунок № 4 – График времени работы при различных значениях числа потоков при одинаковом schedule(dynamic, chunk_size)



Графики на рисунках 5 и 6 показывают время работы программы при одинаковом значении числа потоков при различных параметрах schedule (static и dynamic соответственно).

Рисунок № 5 – График времени работы при одинаковом значении числа потоков при разных schedule(static, chunk_size)

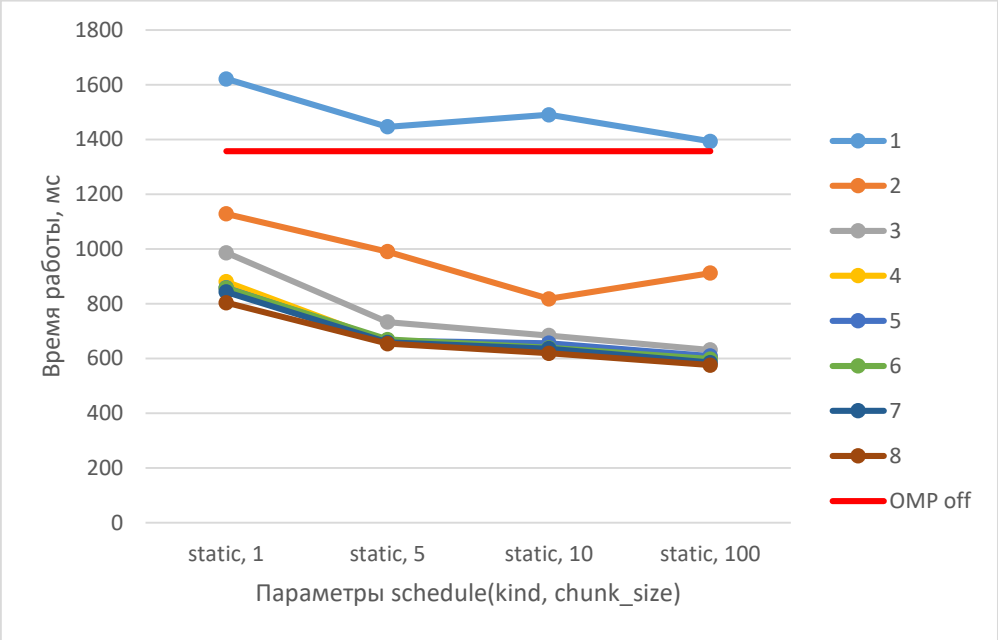
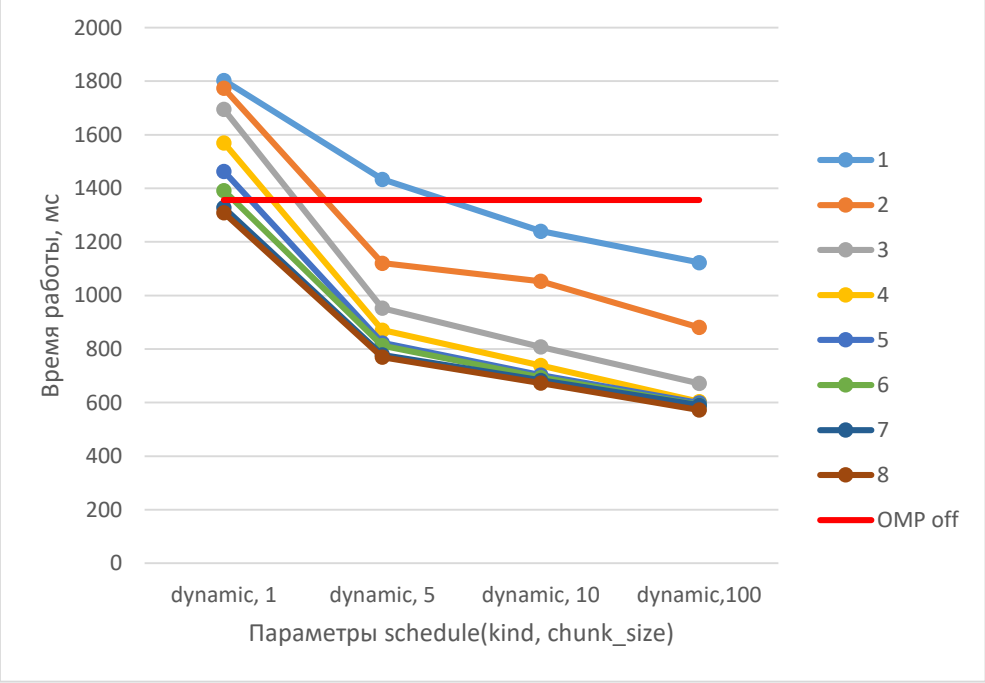


Рисунок № 6 – График времени работы при одинаковом значении числа потоков при разных schedule(dynamic, chunk_size)



Благодаря графикам можно сделать некоторые выводы. При увеличении числа потоков программа начинает работать быстрее. Увеличение порции итераций на поток chunk_size также даёт прирост скорости. При этом скорость в режиме dynamic больше зависит от chunk_size, чем в режиме static, видимо, потому, что потоки берут большие порции итераций и реже прерываются для получения новых. Кроме того, видно, что с одним потоком программа может работать даже медленнее, чем с выключенным OpenMP, так как на создание потока тоже уходят время и ресурсы.

Листинг

Компилятор GCC 11.2.0

main.cpp

```
#include <omp.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <fstream>

using namespace std;

struct pixel {
    int r;
    int g;
    int b;
};

int thread_num;
string input, output;
float factor;
unsigned char magic0, magic1;
int width, height, maxval;
vector<pixel> image;
vector<int> red, green, blue;
vector<int> gray_image;

int main(int argc, char* argv[]) {
    try {
        thread_num = stoi(argv[1]);
        input = argv[2];
        output = argv[3];
        factor = stof(argv[4]);

        ifstream in;
        in.open(input, ios::binary);
        if (!in.is_open()) {
            throw "Cannot open file: " + input;
        }
        ofstream out;
        out.open(output, ios::binary);
```

```
if (!out.is_open()) {
    throw "Cannot open file: " + output;
}
// Reading identification chars
magic0 = in.get();
magic1 = in.get();
out.put(magic0);
out.put(magic1);
out.put(in.get());
if (magic0 != 'P' || magic1 != '5' && magic1 != '6') {
    throw "Incorrect file format";
}
// Reading image width
unsigned char c = in.get();
while (!isspace(c)) {
    out.put(c);
    width = width * 10 + (c - '0');
    c = in.get();
}
out.put(c);
// Reading image height
c = in.get();
while (!isspace(c)) {
    out.put(c);
    height = height * 10 + (c - '0');
    c = in.get();
}
out.put(c);
// Reading color maxvalue
c = in.get();
while (!isspace(c)) {
    out.put(c);
    maxval = maxval * 10 + (c - '0');
    c = in.get();
}
out.put(c);
// Reading image
if (magic1 == '5') {
    // Grayscale
    gray_image.resize(width * height);
    for (int i = 0; i < width * height; i++) {
        gray_image[i] = in.get();
    }
}
else if (magic1 == '6') {
    // RGB
    image.resize(width * height);
    red.resize(width * height);
    green.resize(width * height);
    blue.resize(width * height);
    for (int i = 0; i < width * height; i++) {
        image[i].r = in.get();
        image[i].g = in.get();
        image[i].b = in.get();
        red[i] = image[i].r;
        green[i] = image[i].g;
        blue[i] = image[i].b;
    }
}
double start = omp_get_wtime();
// Finding minimal and maximal values according to factor
int min_value = 0;
```

```
        int max_value = 0;
        int index = (int)(width * height * factor) - 1;
        if (magic1 == '5') {
            // Grayscale
            vector<int> values = gray_image;
            nth_element(values.begin(), values.begin() + index, values.end());
            min_value = values[index];
            nth_element(values.begin(), values.end() - index - 1, values.end());
            max_value = values[values.size() - index - 1];
        }
        else if (magic1 == '6') {
            // RGB
            int min_red, min_green, min_blue;
            int max_red, max_green, max_blue;
            nth_element(red.begin(), red.begin() + index, red.end());
            min_red = red[index];
            nth_element(green.begin(), green.begin() + index, green.end());
            min_green = green[index];
            nth_element(blue.begin(), blue.begin() + index, blue.end());
            min_blue = blue[index];

            min_value = min(min_red, min(min_green, min_blue));
            nth_element(red.begin(), red.end() - index - 1, red.end());
            max_red = red[red.size() - index - 1];
            nth_element(green.begin(), green.end() - index - 1, green.end());
            max_green = green[green.size() - index - 1];
            nth_element(blue.begin(), blue.end() - index - 1, blue.end());
            max_blue = blue[blue.size() - index - 1];
            max_value = max(max_red, max(max_green, max_blue));
        }
        // Calculating new values
        if (magic1 == '5') {
            // Grayscale
            #pragma omp parallel num_threads(thread_num)
            {
                #pragma omp for schedule(runtime)
                for (int i = 0; i < width * height; i++) {
                    gray_image[i] = (gray_image[i] - min_value) * 255 / (max_value -
min_value);
                    if (gray_image[i] < 0) gray_image[i] = 0;
                    else if (gray_image[i] > 255) gray_image[i] = 255;
                }
            }
        }
        else if (magic1 == '6') {
            // RGB
            #pragma omp parallel num_threads(thread_num)
            {
                #pragma omp for schedule(runtime)
                for (int i = 0; i < width * height; i++) {
                    image[i].r = (image[i].r - min_value) * 255 / (max_value -
min_value);
                    image[i].g = (image[i].g - min_value) * 255 / (max_value -
min_value);
                    image[i].b = (image[i].b - min_value) * 255 / (max_value -
min_value);
                    if (image[i].r < 0) image[i].r = 0;
                    if (image[i].g < 0) image[i].g = 0;
                    if (image[i].b < 0) image[i].b = 0;
                    if (image[i].r > 255) image[i].r = 255;
                    if (image[i].g > 255) image[i].g = 255;
                    if (image[i].b > 255) image[i].b = 255;
```

```
                }
            }
        }
        double end = omp_get_wtime();
        // Writing new image
        if (magic1 == '5') {
            for (int i = 0; i < width * height; i++) {
                out.put((char)gray_image[i]);
            }
        }
        else if (magic1 == '6') {
            for (int i = 0; i < width * height; i++) {
                out.put((char)image[i].r);
                out.put((char)image[i].g);
                out.put((char)image[i].b);
            }
        }
        in.close();
        out.close();
        printf("Time (%i thread(s)): %g ms \n", thread_num, (end - start) * 1000);
    }
    catch (string e) {
        cerr << e << '\n';
    }
    return 0;
}
```