САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

**«OPENMP»**

Выполнил: Пруидзе Роман Спартакович

Номер ИСУ: 332986

Cтуд. гр. M3134

Санкт-Петербург

2022

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

**Инструментарий и требования к работе:** C/C++, стандарт OpenMP 2.0.

**Теоретическая часть**

**OpenMP и инструменты, использованные в работе**

Одним из применений **параллельного программирования** является уменьшение времени выполнения последовательных программ. Когда команды распределяются по потокам и выполняются параллельно. Существуют различные параллельные архитектуры с общей памятью и распределенной. OpenMP предназначен для систем с общей памятью.

**OpenMP** – открытый стандарт для создания параллельных программ. Этот инструмент позволяет создавать многопоточные приложения на многопроцессорных системах с общей памятью. То есть памятью, которая является общим ресурсом для каждого ядра, и они имеют полный доступ ко всему адресному пространству. Данный стандарт описывает набор директив компилятора, библиотечных функций и переменных окружения.

Основным понятием OpenMP является **параллельный регион.** Именно в месте, где он задан программа исполняется параллельно. Задаются такие места с помощью директивы **#pragma omp parallel{}**. Как только компилятор встречает ее, он вставляет инструкции для создания параллельных потоков. Каждый поток имеет свой уникальный номер. Получить данный номер мы можем посредством функции **omp\_get\_thread\_num()**. Функция **omp\_set\_num\_threads()** позволяет задать количество потоков. Функция **omp\_get\_max\_threads()** позволяет узнать максимальное количество потоков на данном компьютере. Чтобы распараллелить цикл нам необходимо использовать директиву **#pragma omp parallel for**, а также указать аргументы: **shared(\*variables\*)** – данное условие указывает на то, что все перечисленные в нем переменные будут разделяться между потоками и к ним будет открыт доступ в параллельной секции, **default(none)** – данное условие определяет область видимости переменных внутри параллельной области, где **none** – указывает на то, что все переменные должны быть явно указаны, **schedule(type[, chunk\_size]),** где в качестве **chunk\_size** указывается размер порции данных, которые будут последовательно подгружать в освободившиеся потоки. Параметр **type** – определяет тип загрузки.

Рассмотрим два типа загрузки: динамическую и статическую. При динамической количество потоков в программе может варьироваться от одной области параллельного выполнения к другой, и статическая, когда количество потоков фиксировано.

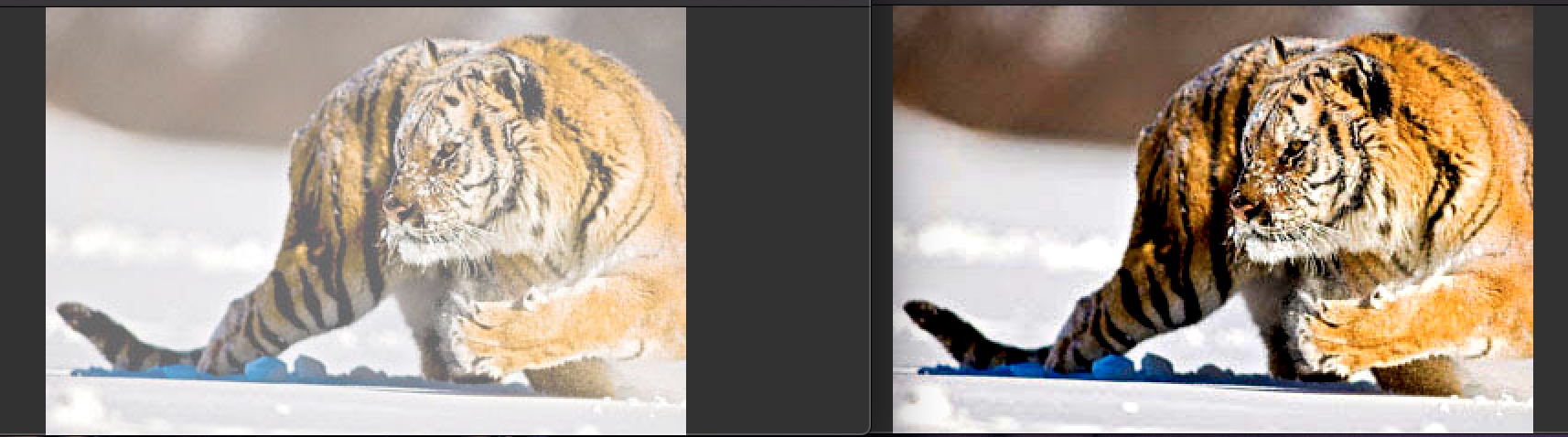
* При **static**: В этом случае вся совокупность загружаемых процессов разбивается на равные порции размера **chunk\_size**, и эти порции последовательно распределяются между потокамиc с первого по последний.
* При **dynamic:** В этом случае вся совокупность загружаемых процессов, как и в предыдущем варианте, разбивается на равные порции размера **chunk\_size**, но эти порции загружаются последовательно в освободившиеся потоки.

Когда мы говорим о параллельном программировании стоит всегда в первую очередь учитывать **race condition**. Это ошибка проектирования многопоточных приложений, которая возникает, когда потоки обращаются к одной и той же области памяти. Вследствие этой ошибки, например, в циклах, чьи итерации распределяются по потокам может возникнуть ошибка, при пересчете одной и той же переменной. Это приводит к тому, что мы будем получать некорректный результат. Методы борьбы могут быть различные. Как простые программные, когда мы просто придумываем алгоритм, который будет вести вычисления независимо для каждого из потоков, тем самым ограничивая доступ к этой области для остальных, либо используя средства синхронизации OpenMP. Средство, которое используется в этой работе – **critical** (критическая секция). Наличие критической секции в параллельном блоке гарантирует, что она в каждый конкретный момент времени будет выполняться только одним потоком. Т.е. когда один поток находится в критической секции, все остальные потоки, которые готовы в нее войти, находятся в приостановленном состоянии. Это гарантирует избегание **race condition**.

**Описание реализуемого алгоритма**

В работе будет описана HARD – модификация. Эта модификация подразумевает написание алгоритма для повышения контрастности изображения без изменения оттенков. Контраст изображения – это отношение яркостей самой светлой и самой темной частей. Для изменения контрастности изображения необходимо рассчитать гистограмму – распределение пикселей по значению яркости в диапазоне [0;255;]. Далее необходимо «растянуть» гистограмму, то есть сопоставить самую низкую и самую высокое значение яркости с минимальное и максимальное значение полного диапазона [0;255]. При этом необходимо учитывать процент самых ярких и тусклых пикселей на гистограмме исключить, чтобы не было шума, данный процент мы задаем в аргументах командной строки в виде коэффициента. Далее мы просчитываем новое значение яркости для каждого пикселя по формуле контрастности:

где , – значения яркости каждого пикселя, минимальное и максимальное значение яркости соответственно. Далее новые просчитанные значения записываются в новый файл. На выходе получается изображение с более высокой контрастностью, которая зависит от ранее заданного коэффициента. Результат работы такого алгоритма представлен на Рисунке №1.

Рисунок №1 Изображение до и после применения алгоритма.

**Практическая работа**

В начале программа считывает файл. По информации, которая находится в первых трех строках мы узнаем максимальное значение яркости, размер, а также какого формата это изображение в зависимости от числа после P (P5 – pgm и P6 – ppm). Далее определяем количество каналов в зависимости от формата. Чтобы составить гистограмму нам необходимо посчитать сколько пикселей определенной яркости у нас есть. Счетчиками выступают элементы массива длиной 256, где каждому индексу соответствует определённое значение яркости. Для трехканальных изображений мы просчитываем отдельно 3 гистограммы. И идем по массиву с шагом 3 в целях экономии времени исполнения. После подсчета и создания гистограмм мы определяем максимальное и минимальное значение яркости(в случае 3-каналов берем самое маленькое и самое большое из 6 значений), отбрасывая определенный процент пикселей слева и справа по шкале, чтобы исключить шум. Эти значения понадобятся нам, чтобы посчитать новые для каждого пикселя. Вычислять для каждого пикселя – нецелесообразно. Намного более практично сделать предподсчёт для 256 значений яркости и обращаться к массиву при формировании нового файла. Новые значения считаются по формуле автоматической контрастности, которая описана в разделе описания алгоритма. Далее мы формируем новый файл, предварительно указав его формат, а также размер и максимальное значение. Но так как мы хотим оптимизировать время выполнения программы, то мы используем директивы компилятора стандарта OpenMP. Выделяя места, которые должны выполняться параллельно. Таких мест в своей программе я нашел два, это формирование нового файла и расчёт гистограммы. Именно в этих местах находятся циклы, с большим количеством итераций, которые нуждаются в оптимизации. Циклы, отвечающие за предподсчет и нахождение минимумов и максимумов, не нуждаются в этом, так как работают только на 256 значениях и их время выполнения незначительно. Также в критическую секцию мы вставляем часть кода первого цикла, где идет обращению к массиву с гистограммой. Так как это общее адресное пространство всех потоков, а следовательно оно не застраховано от **race condition**. Второй цикл в этом не нуждается, так как там переменные, с которыми работают отдельные потоки не пересекаются.

**Графики времени работы программы**

Рисунок №2 и Рисунок №4 демонстрируют время выполнения программы на .ppm изображении размером 600Мб. Рисунок №3 – на изображении размером 32Мб. По технической причине данные для этих графиков были получены на разных компьютерах, но так как на разном оборудовании мы имеем разное время выполнения, то нам важны значения относительно друг друга в рамках одного эксперимента. Следовательно, на чистоту эксперимента это повлиять не должно.

Каждое полученное значение является средним арифметическим нескольких 4-х замеров.

Рисунок №2 — График зависимости времени выполнения программы от количества тредов.

Рисунок №3 — График зависимости времени выполнения программы от размера порции данных и типа загрузки процессов.

Рисунок №4 — Время выполнения программы при 1 потоке с OpenMP и без.

Из графиков видно, что при большой порции данных и одном количестве потоков разница не является значимой. Аналогично разница между программы с OpenMP и без него при 1 потоке. Из графика 1 очевидна разница: при увеличении количества тредов время вычислений уменьшается значительно.

**Листинг кода**

**HW5\_PRUIDZE.cpp**

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <algorithm>

#include <omp.h>

#include <cmath>

using namespace std;

int main(int argc, char\* argv[]) {

unsigned int threads = std::stoi(argv[1]);

if (threads == 0) threads = omp\_get\_max\_threads();

omp\_set\_num\_threads(threads);

std::string file\_name = argv[2];

std::string output\_file\_name = argv[3];

double k = std::stod(argv[4]);

unsigned char header[2];

unsigned int width, height, max\_v;

std::ifstream in(file\_name, std::ios::binary);

std::ofstream out(output\_file\_name, std::ios::binary);

if (out.fail()) {

std::cout << "Permission denied or wrong path";

exit(1);

}

if (in.fail()) {

std::cout << "File not found";

exit(1);

}

in >> header[0] >> header[1];

out << header[0] << header[1] << '\n';

if (header[0] != 'P' || (header[1] != '5' && header[1] != '6')) {

std::cout << "Invalid file" << '\n';

exit(1);

}

in >> width >> height >> max\_v;

out << width << ' ' << height << '\n' << max\_v << '\n';

int size = width \* height;

int size\_channel = size;

if (header[1] == '6') size \*= 3;

std::vector<unsigned char> arr(size);

in.get();

in.read(reinterpret\_cast<char\*>(&arr[0]), size);

in.close();

double time = omp\_get\_wtime();

const int channels = header[1] == '5' ? 1 : 3;

std::vector<std::vector<unsigned int>> values\_br\_result(channels, std::vector<unsigned int>(256));

#pragma omp parallel shared(values\_br\_result, arr, size, cout, channels) default(none)

{

unsigned int arr\_thread[3][256];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 256; j++) {

arr\_thread[i][j] = 0;

}

}

#pragma omp for schedule(static)

for (int i = 0; i < size; i += channels) {

arr\_thread[0][arr[i]]++;

if (channels == 3) {

arr\_thread[1][arr[i + 1]]++;

arr\_thread[2][arr[i + 2]]++;

}

}

#pragma omp critical

{

for (int p = 0; p < values\_br\_result.size(); p++)

for (int k = 0; k < 256; k++)

values\_br\_result[p][k] += arr\_thread[p][k];

}

}

unsigned char min\_value = 255;

unsigned char max\_value = 0;

int up\_bound = size\_channel \* (1 - k);

int low\_bound = k \* size\_channel;

for (int i = 0; i < values\_br\_result.size(); i++) {

int sum = 0;

for (int j = 0; j < 256; j++) {

sum += values\_br\_result[i][j];

if (sum > low\_bound) {

min\_value = std::min(min\_value, (unsigned char)j);

}

if (sum < up\_bound) {

max\_value = std::max(max\_value, (unsigned char)j);

}

}

}

max\_value++;

std::vector<unsigned char> result\_values(256);

for (int i = 0; i < 256; i++) {

int mor = round(255.0 \* (i - min\_value) / (max\_value - min\_value));

result\_values[i] = mor < 0 ? 0 : (mor > 255 ? 255 : mor);

}

#pragma omp parallel for shared(arr, result\_values) default(none) schedule(dynamic, 65536)

for (int i = 0; i < arr.size(); i++) arr[i] = result\_values[arr[i]];

std::cout << "Time(" << threads << " thread(s)): " << (omp\_get\_wtime() - time) \* 1000 << " ms" << std::endl;

out.write(reinterpret\_cast<char\*>(&arr[0]), size);

out.close();

}