САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

«OpenMP»

Выполнил(а): Тропин Михаил Алексеевич

Номер ИСУ: 334842

студ. гр. М3139

Санкт-Петербург

2021

Цель работы: знакомство со стандартом OpenMP.

Инструментарий и требования к работе: работа выполнена на C++. Стандарт OpenMP 2.0.

Теоретическая часть

OpenMP – это библиотека для C и C++ для параллельного программирования с общей памятью. Потоки в OpenMP создаются в одном процессе и имеют свою локальную память, но также, все потоки имеют доступ к памяти процессора.

Директива #pragma omp parallel задает параллельную область(порождает несколько потоков). Число потоков можно указать аргументом num_threads(x), по умолчанию количество потоков будет соответствовать количеству вычислительных ядер машины. При этом общие переменные для параллельных потоков указываются в параметре shared(...).

Чаще всего используется для параллельных циклов. Опция schedule позволяет контролировать распределение итераций между потоками.

- schedule(static): Статическое планирование. Количество итераций будет равно распределено между потоками.
- schedule(static, x): каждый поток в начале получает x итераций, затем продолжает планирование, пока есть итерации.
- schedule(dynamic, x): динамическое планирование. если x не указано, считается равным 1. Каждый поток получает x итераций, выполняет их, после этого запрашивает новые.

Практическая часть

Решение задачи hard:

Считываю данные с файла, сохраняю в структуру image. Пиксели храню в одномерном массиве data этой структуры.

Для нормализации картинки нахожу сначала для каждого цвета пиксель с минимальным и максимальным значением с учетом коэффициента: нужно найти k-ый пиксель по возрастанию и по убыванию, для этого воспользовался сортировкой подсчетом. k = количество пикселей одного цвета * коэффициент.

Для сортировки подсчетом нужно узнать для каждого значения сколько таких пикселей есть на картинке. Воспользуюсь параллельным циклом для ускорения.

Затем параллельным циклом перебираю все пиксели. Новое значение ищу так: для каждого цвета нахожу коэффициент x_r(например для цвета r), на который нужно домножить текущее значение пикселя, чтобы значение r*x было примерно равно (r - mn_r) * 255 / (mx_r - mn_r), где r – значение текущего пикселя цвета r. mn_r, mx_r – соответствующие минимальные и максимальные значения. Затем нахожу истинный коэффициент x, на который домножаю значения пикселей, таким образом оттенок не изменяется.

Графики

График 1(рис. 1) параметры: schedule(static), коэффициент = 0.2

левая колонка – время в мс, снизу – количество потоков.

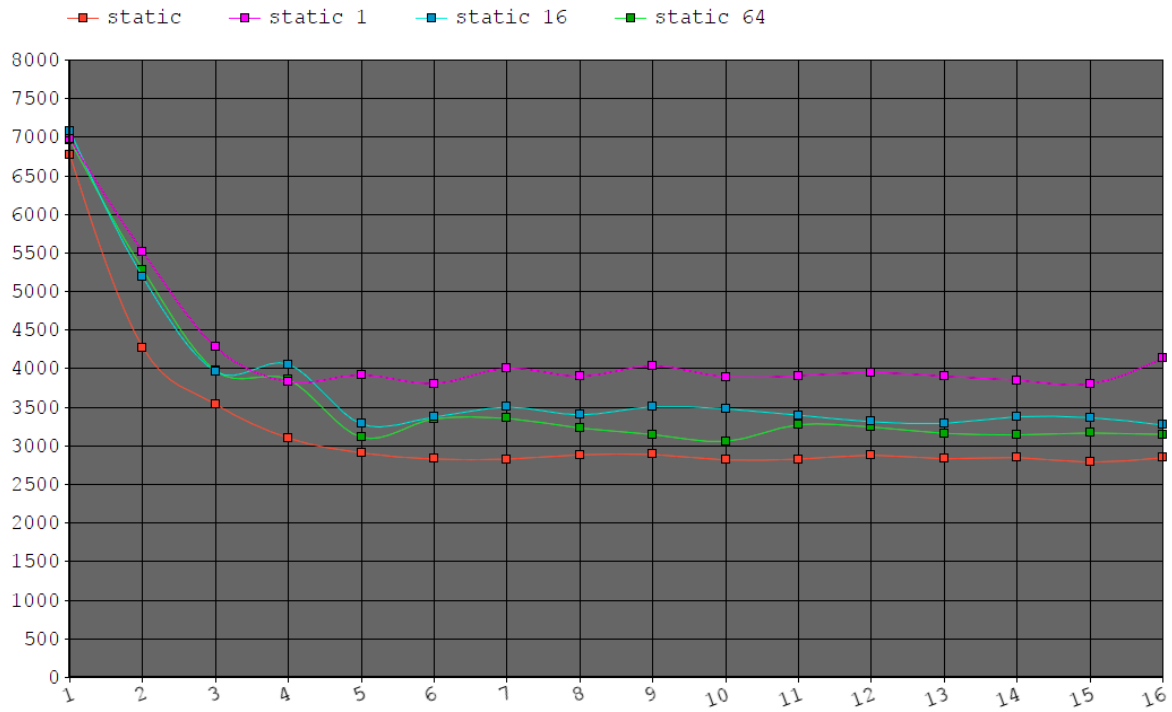

рисунок 1 – график для static

График 2(рис. 2) параметры: 1 поток и коэффициент = 0.2

левая колонка – время в мс, снизу – параметры. (brute – запуск без openMP)

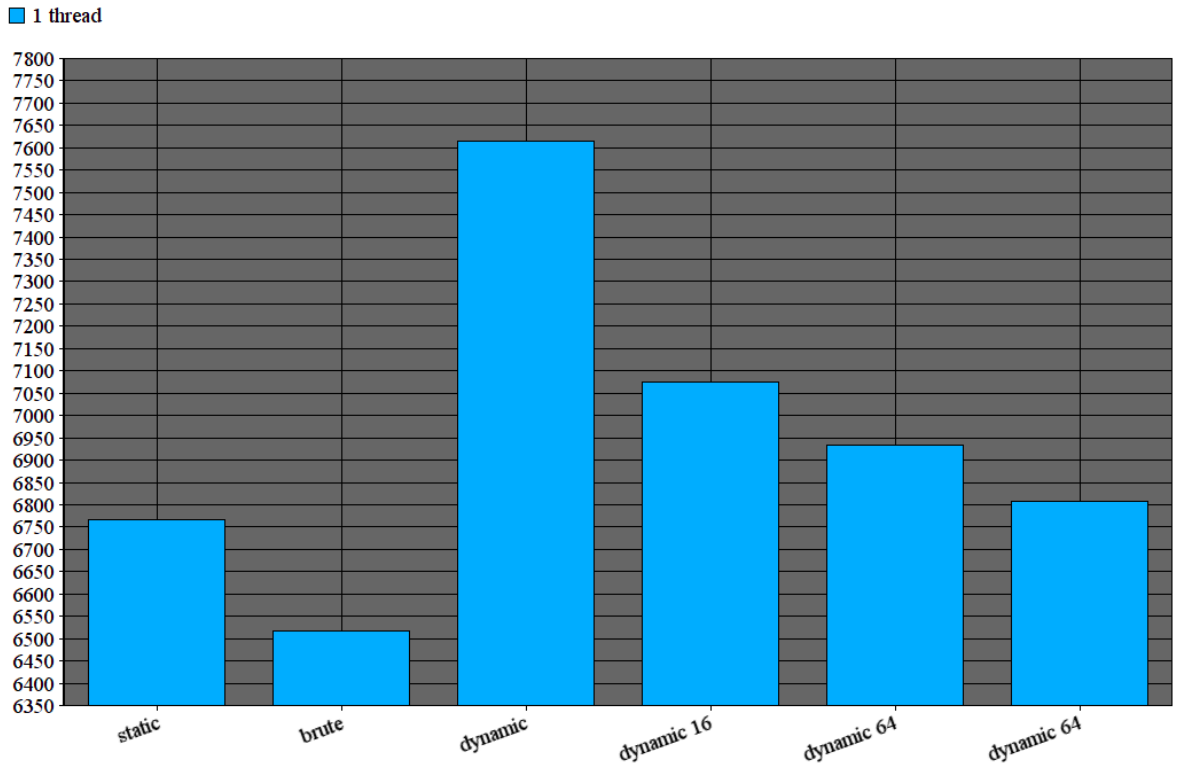

рисунок 2 – график для запусков с 1 потоком

Листинг

Компилировал: `g++ -fopenmp -std=c++17 -Wall -Wextra -Wconversion hw5.cpp -o hw5.exe`

hw5.cpp

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

template <class T> bool ckmin(T &a, T b) {return a > b ? a=b, true : false;}
template <class T> bool ckmax(T &a, T b) {return a < b ? a=b, true : false;}


void my_assert(bool exp, std::string assert_message) {
   if (exp) return;
      std::cerr << "Assert failed: " << std::endl << assert_message << std::endl;
   abort();
}

bool is_digit(char c) {
   return c >= '0' && c <= '9';
}

bool is_positive_integer(std::string &s) { // includes 0
   for (size_t i = 0; i < s.size(); i++) {
       if (!is_digit(s[i])) return false;
   }
   int res = stoi(s);
   return s == std::to_string(res);
}
```

```cpp
bool is_integer(std::string &s) {
    if (s[0] == '-') {
        std::string t = s.substr(1, int(s.size()) - 1);
        return is_positive_integer(t);
    }
    return is_positive_integer(s);
}

bool is_coef(std::string &s) {
    int dot = -1;
    for (size_t i = 0; i < s.size(); i++) {
        if (s[i] == '.') {
            if (dot == -1) dot = int(i);
            else return false;
        } else if (!is_digit(s[i])) {
            return false;
        }
    }
    float res = stof(s);
    return res >= 0.0 && res < 0.5;
}

bool is_ppm_file(std::string &s) {
    int sz_s = int(s.size());
    if (sz_s < 4) return false;
    std::string suffix = s.substr(sz_s - 4, 4);
    return suffix == ".ppm";
}

bool is_pgm_file(std::string &s) {
    int sz_s = int(s.size());
    if (sz_s < 4) return false;
    std::string suffix = s.substr(sz_s - 4, 4);
    return suffix == ".pgm";
}

bool is_file_exist(std::string &s) {
    std::ifstream f(s.c_str());
    return f.good();
}
```

```
int read_int(std::ifstream &in) {
    std::string s;
    my_assert(bool(in >> s), "Expected integer but found end of file");
    my_assert(is_integer(s), s + " expected as integer");
    return stoi(s);
}

int read_positive_int(std::ifstream &in) {
    int res = read_int(in);
    std::string s = std::to_string(res);
    my_assert(is_positive_integer(s), s + " expected as positive integer");
    return res;
}

void check_end_of_file(std::ifstream &in) {
    std::string s;
    bool res = !(bool(in >> s));
    my_assert(res, "Expected end of file, but found " + s);
    return ;
}

int threads;


struct image {
    int type; //[1] -- gray, [3] -- rgb
    unsigned int width;
    unsigned int height;
    size_t siz;
    float coef;

    unsigned char *data;


    void fit() {
        siz = width * height * type;
        data = new unsigned char[siz];
    }
```

```
    inline unsigned char get(int x,int y) {
        return data[x * height + y];
    }

    inline unsigned char get(int x,int y,int c) { //c = 0: r, c = 1: g, c = 2:
b
        return data[(x * height + y) * 3 + c];
    }

    inline unsigned char get(int x) {
        return data[x];
    }
} img;



void read_img(char *file) {
    FILE *in = fopen(file, "rb");
    if (img.type == 3) {
          my_assert(fscanf(in, "P6\n%d %d\n255\n", &img.width, &img.height) ==
2, "Input file has invalid parametrs");
    } else {
          my_assert(fscanf(in, "P5\n%d %d\n255\n", &img.width, &img.height) ==
2, "Input file has invalid parametrs");
    }

    img.fit();

    size_t to_read = sizeof(unsigned char[img.siz]);
    size_t read = fread(img.data, 1, to_read + 1, in);
    my_assert(!ferror(in) && read == to_read, "Reading file error");
    my_assert(feof(in), "End of file expected");
    fclose(in);
}

inline int min(int a,int b) {
    return (a < b ? a : b);
}
```

```
inline int max(int a,int b) {
   return (a > b ? a : b);
}

inline int norm_value(double x) {
   return max(0, min(255, int(round(x))));
}

void normalize_gray() {
   unsigned int cnt[256];
   unsigned int kth = (int)round((float)img.siz * img.coef);

#pragma omp parallel for schedule(static) shared(cnt) num_threads(threads)
   for (size_t i = 0; i < 256; i++) {
       cnt[i] = 0;
   }

#pragma    omp    parallel    for    schedule(static)    shared(img,    cnt)
num_threads(threads)
   for (unsigned int d = 0; d < img.siz; d++) {
       cnt[img.get(d)]++;
   }

   unsigned int k = kth;
   int mn = 255, mx = 0;
   for (int i = 0; i < 256; i++) if(cnt[i] > 0) {
       if (k > cnt[i]) k -= cnt[i];
       else {
           mn = i;
           break;
       }
   }

   k = kth;
   for (int i = 255; i >= 0; i--) if(cnt[i] > 0) {
       if (k > cnt[i]) k -= cnt[i];
       else {
           mx = i;
           break;
       }
```

```
    }
    if (mx <= mn) return;

#pragma omp parallel for schedule(static) shared(img, mn, mx)
    for (unsigned int d = 0; d < img.siz; d++) {
          img.data[d] = (unsigned char)norm_value((img.get(d) - mn) * 255.0 /
(mx - mn));
    }
}

struct pixel{
    int r, g, b;

    pixel(){
        r = g = b = 0;
    }

    pixel(int _r, int _g, int _b) {
        r = _r;
        g = _g;
        b = _b;
    }
};

void normalize_rgb() {
    pixel cnt[256];
    unsigned int kth = (int)floor((float)(img.height * img.width) * img.coef);

#pragma omp parallel for schedule(static) shared(img, cnt)
num_threads(threads)
    for (unsigned int d = 0; d < img.height * img.width; d++) {
        cnt[img.get(d*3+0)].r++;
        cnt[img.get(d*3+1)].g++;
        cnt[img.get(d*3+2)].b++;
    }

    pixel k(kth, kth, kth);
    pixel mn(0,0,0), mx(255,255,255);

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

```
        if (k.r >= 0) mn.r = i;
        k.r -= cnt[i].r;

        if (k.g >= 0) mn.g = i;
        k.g -= cnt[i].g;

        if (k.b >= 0) mn.b = i;
        k.b -= cnt[i].b;
    }

    k = pixel(kth, kth, kth);
    for (int i = 255; i >= 0; i--) {
        k.r -= cnt[i].r;
        if (k.r >= 0) mx.r = i;

        k.g -= cnt[i].g;
        if (k.g >= 0) mx.g = i;

        k.b -= cnt[i].b;
        if (k.b >= 0) mx.b = i;
    }

    // std::cerr << mn.r << ' ' << mn.g << ' ' << mn.b << std::endl;
    // std::cerr << mx.r << ' ' << mx.g << ' ' << mx.b << std::endl;
    if (mx.r <= mn.r || mx.g <= mn.g || mx.b <= mn.b) return;


#pragma omp parallel for schedule(static) shared(img, mn, mx) num_threads(threads)
    for (unsigned int d = 0; d < img.width * img.height; d++) {
            unsigned char r = img.get(d*3+0), g = img.get(d*3+1), b = img.get(d*3+2);

          double xr = r == 0 ? 0 : norm_value((r - mn.r) * 255.0 / (mx.r - mn.r))*1. / r;
          double xg = g == 0 ? 0 : norm_value((g - mn.g) * 255.0 / (mx.g - mn.g))*1. / g;
          double xb = b == 0 ? 0 : norm_value((b - mn.b) * 255.0 / (mx.b - mn.b))*1. / b;
```

```
        double x = (xr + xg + xb) / 3;

        img.data[d * 3 + 0] = (unsigned char)norm_value(r * x);
        img.data[d * 3 + 1] = (unsigned char)norm_value(g * x);
        img.data[d * 3 + 2] = (unsigned char)norm_value(b * x);
    }
}


void write_img(char *file) {
    FILE *out = fopen(file, "wb");
    if (img.type == 1) {
        fprintf(out, "P5\n%d %d\n255\n", img.width, img.height);
    } else {
        fprintf(out, "P6\n%d %d\n255\n", img.width, img.height);
    }

    fwrite(img.data, img.siz, 1, out);
    fclose(out);
}

int main(int argc, char *argv[]) {
    std::vector<std::string> args(argv, argv + argc);
    // {hw5.exe, [threads], [input_file], [output_file], [coef]}
    my_assert(int(args.size()) == 5, "Expected 4 arguments");
     my_assert(is_positive_integer(args[1]), " Expected " + args[1] + " as
positive integer");
     my_assert(is_file_exist(args[2]), "Input file " + args[2] + " doesn't
exist");
     my_assert(is_ppm_file(args[2]) || is_pgm_file(args[2]), "Input file: " +
args[2] + " expected as .pnm or .pgm file");
     my_assert(is_ppm_file(args[3]) || is_pgm_file(args[3]), "Output file: " +
args[3] + " expected as .pnm or .pgm file");
     my_assert(is_ppm_file(args[2]) == is_ppm_file(args[3]), "Input and Output
file must have the same type");
    my_assert(is_coef(args[4]), "Coefficient expected as float number in [0.0,
0.5)");

    threads = stoi(args[1]);
    img.coef = stof(args[4]);
```

```
    img.type = is_ppm_file(args[2]) ? 3 : 1;

    read_img(argv[2]);

    double start_t = omp_get_wtime();
    if (img.type == 1) {
        normalize_gray();
    }else {
        normalize_rgb();
    }
    double end_t = omp_get_wtime();

     printf("Time (%d thread(s)): %f ms\n", threads, float(end_t - start_t) *
1000);

    write_img(argv[3]);

    return 0;
}
```