Лабораторная работа №4	M3136	2022
OpenMp	Шибанов Игорь Александрович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: работа выполнена в Visual Studio 2022. Язык - C++, компилятор - MSVC.

Описание: Пороговая фильтрация изображения методом Оцу. Необходимо реализовать алгоритм для трёх порогов.

Вариант: Hard

Описание конструкций OpenMP

OpenMP — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Конструкция #pragma в языке Си/Си++ используется для задания дополнительных указаний компилятору. С помощью этих конструкций можно указать как осуществлять выравнивание данных в структурах, запретить выдавать определенные предупреждения и так далее.

Использование специальной директивы **omp** указывает на то, что команды относятся к OpenMp. #pragma omp…

Директива **parallel**

Директива parallel указывает, что структурный блок кода должен быть выполнен параллельно в несколько потоков. Каждый из созданных потоков выполнит одинаковый код, содержащийся в блоке, но не одинаковый набор команд. В разных потоках могут выполняться различные ветви или обрабатываться различные данные

#pragma omp parallel

Директива **for**

Вызывает разделение работы в for цикле внутри параллельного региона между потоками.

#pragma omp parallel for

Директива **schedule**

Для распределения работы между процессами в OpenMP имеется директива schedule с параметрами, позволяющими задавать различные режимы загрузки процессоров. Ниже приведен общий вид предложения schedule в OpenMP.

schedule(type [, chunk])

Загрузка типа **static**

В этом случае вся совокупность загружаемых процессов разбивается на равные порции размера chunk, и эти порции последовательно распределяются между процессорами (или потоками, которые затем и выполняются на этих процессорах) с первого до последнего и т. д.

Загрузка типа **dynamic**

В этом случае вся совокупность загружаемых процессов, как и в предыдущем варианте, разбивается на равные порции размера chunk, но эти порции загружаются последовательно в освободившиеся потоки (процессоры).

Директива **critical**

Этот тип синхронизации используется для описания структурных блоков, выполняющихся только в одном потоке из всего набора параллельных потоков.

#pragma omp critical

Изменить количество потоков можно командой omp_set_num_threads(n) или с помощью директивы num_threads(n).

Описание работы написанного кода

Основные формулы:

$$p(f) = n_f / N$$

$$q_1(f_T) = \sum_{f=1}^{f_T} p(f)$$

$$\mu_1(f_T) = \sum_{f=1}^{f_T} \frac{fp(f)}{q_1(f_T)}$$

Для этих формул написан предподсчет (функция recount). С помощью префиксных сумм можно найти q и mu (getQ и getMu).

В функции getBestF перебираются все возможные пороги и считается сигма для них. Формула для сигмы:

$$\sigma_b^2 = \sum_{k=1}^{M} q_k {\mu_k}^2$$

Пороги с наибольшей сигмой будут искомыми. Также если q = 0, то такие пороги будут пропущены, т.к на 0 делить нельзя.

В функции readFile считываются данные из файла и проверяется их валидность.

Функция getHistogram считает гистограмму по входным данным.

Функции convert возвращает новый цвет, исходя из порогов.

Функция convertImg конвертирует старые цвета в новые.

Функция outFile выводит результат в нужный файл.

Распараллеливание

Заметим, что функцию getBestF можно распараллелить (новая функция getBestFWithOmp). С помощью #pragma omp parallel for паралеллим 3 фора, которые считают пороги. Также в данной функции есть команда изменения количества потоков (omp_set_num_threads), она срабатывает при нужных входных данных. Еще прописана директива schedule. Т.к в форах есть if, в который могут зайти сразу несколько потоков и изменить значение наилучшей сигмы, я прописал директиву critical. Чтобы код исполнялся быстрее, сначала прописан if, потом critical, потом такой же if.

```
if (sigma > bestSigma) {
#pragma omp critical
    if (sigma > bestSigma) {
        bestSigma = sigma;
        bestF[0] = f1;
        bestF[1] = f2;
```

```
                bestF[2] = f3;
            }
        }
```

Если убрать первый if, то потоки будут ждать один поток, который работает с ифом. Это увеличит время работы программы.

Также можно распараллелить функцию convertImg, т.к она перебирает все цвета, несвязанные между собой.

Результат работы программы

Процессор AMD Ryzen 7 4800H

77 130 187

Time (0 thread(s)): 6.45399 ms

Экспериментальная часть

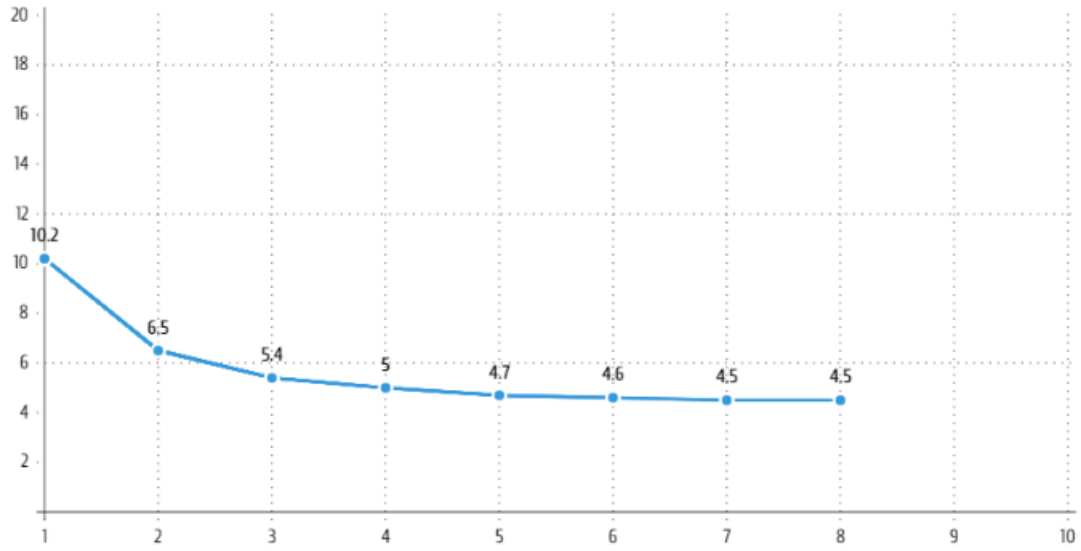

Рисунок 1 – график времени работы программы при разном количестве

потоков и одинаковом параметре schedule. Ось Ox – количество потоков, Oy – время в мс.

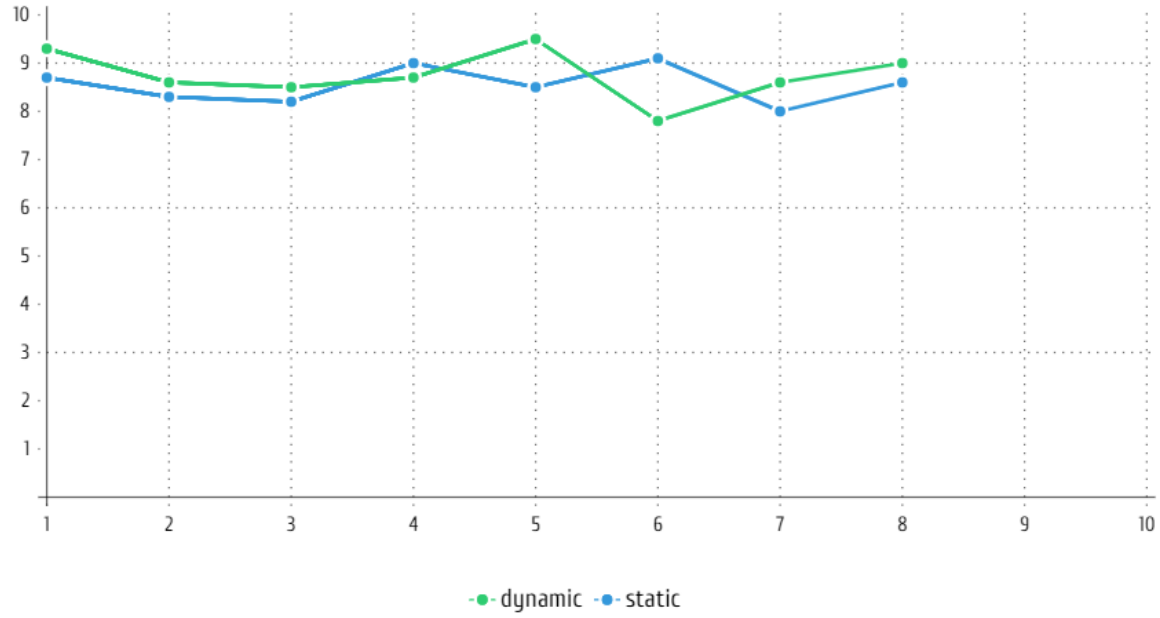

Рисунок 2 – график времени работы программы при одинаковом количестве потоков и при различных параметрах schedule. Ось Ox – параметр chunk_size, Oy – время в мс.

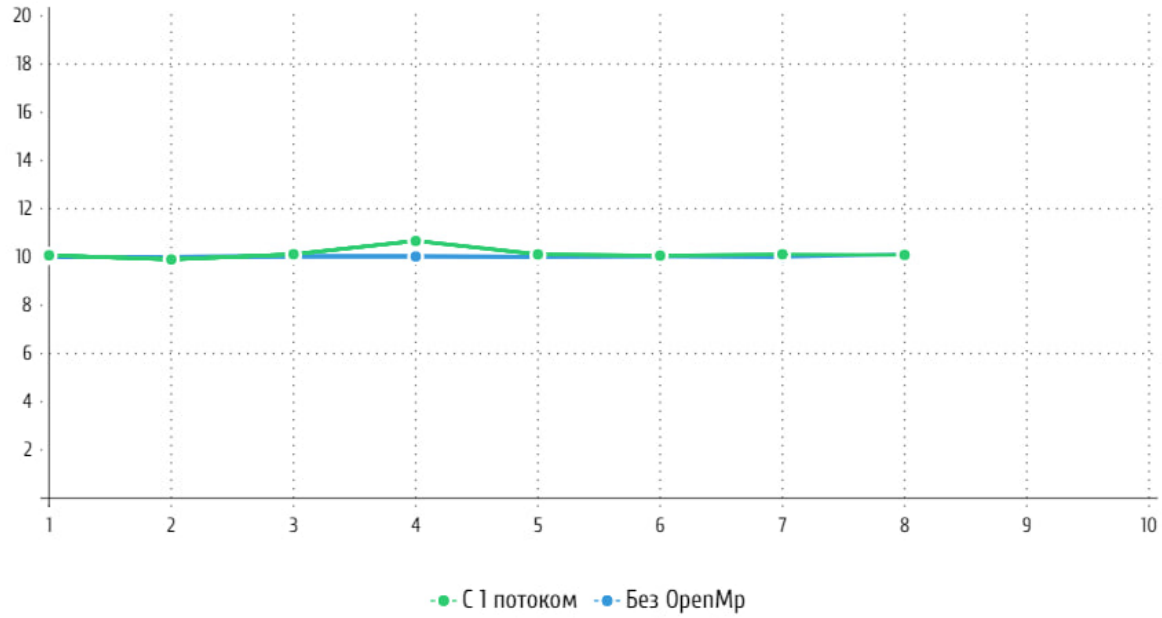

Рисунок 3 – график времени работы программы с выключенным openmp и с включенным с 1 потоком (schedule- kind принимает значение dynamic, chunk_size – 1 и несколько других значений). Ось Ox – номер запуска, ось Oy – время в мс.

Источники

https://intuit.ru/studies/courses/1112/232/lecture/6025?page=2

https://ru.wikipedia.org/wiki/OpenMP

https://learn.microsoft.com/ru-ru/cpp/parallel/openmp/1-introduction?view=msvc-170

https://youtu.be/_MKbLk6K_Tk

Листинг кода

hard.cpp

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <omp.h>
#include <stdexcept>
using namespace std;

string version;
int width;
int height;
int colorsNumber;
int histogram[256];
double p[256];
double q[256];
double mu[256];
int threads = 0;
vector<unsigned char> img;

void recount() {
    for (int f = 0; f < 256; f++) {
        p[f] = double(histogram[f]) / (width * height);
```

```
        if (f != 0) {
            q[f] = q[f - 1];
            mu[f] = mu[f - 1];
        }
        q[f] += p[f];
        mu[f] += f * p[f];
    }
}

double getP(int f) {
    return p[f];
}

double getQ(int startf, int finishf) {
    if (startf == 0) {
        return q[finishf];
    }
    return q[finishf] - q[startf - 1];
}

double getMu(int startf, int finishf) {
    double qCopy = getQ(startf, finishf);
    if (qCopy == 0) {
        return -1;
    }
    double sum = mu[finishf];
    if (startf != 0) {
        sum = sum - mu[startf - 1];
    }
    return sum / qCopy;
}


double getSigma(int startf, int finishf) {
    double copyMu = getMu(startf, finishf);
    if (copyMu == -1) {
        return -1;
    }
    return  copyMu * copyMu * getQ(startf, finishf);
}

void getBestFWithOmp(int bestF[3]) {
    double bestSigma = 0;
    if (threads != 0) {
        omp_set_num_threads(threads);
    }
#pragma omp parallel for schedule(dynamic)
    for (int f1 = 0; f1 <= colorsNumber; f1++) {
        double sigma1 = getSigma(0, f1);
```

```
        if (sigma1 == -1) {
            continue;
        }
        for (int f2 = f1 + 1; f2 <= colorsNumber; f2++) {
            double sigma2 = getSigma(f1 + 1, f2);
            if (sigma2 == -1) {
                continue;
            }
            for (int f3 = f2 + 1; f3 <= colorsNumber; f3++) {
                double sigma3 = getSigma(f2 + 1, f3);
                double sigma4 = getSigma(f3 + 1, colorsNumber);
                if (sigma3 == -1 || sigma4 == -1) {
                    continue;
                }
                double sigma = sigma1 + sigma2 + sigma3 + sigma4;
                if (sigma > bestSigma) {
#pragma omp critical
                    {
                        if (sigma > bestSigma) {
                            bestSigma = sigma;
                            bestF[0] = f1;
                            bestF[1] = f2;
                            bestF[2] = f3;
                        }
                    }
                }
            }
        }
    }
}

void getBestF(int bestF[3]) {
    if (threads != -1) {
        getBestFWithOmp(bestF);
        return;
    }
    double bestSigma = 0;
    for (int f1 = 0; f1 <= colorsNumber; f1++) {
        double sigma1 = getSigma(0, f1);
        if (sigma1 == -1) {
            continue;
        }
        for (int f2 = f1 + 1; f2 <= colorsNumber; f2++) {
            double sigma2 = getSigma(f1 + 1, f2);
            if (sigma2 == -1) {
                continue;
            }
            for (int f3 = f2 + 1; f3 <= colorsNumber; f3++) {
```

```
                double sigma3 = getSigma(f2 + 1, f3);
                double sigma4 = getSigma(f3 + 1, colorsNumber);
                if (sigma3 == -1 || sigma4 == -1) {
                    continue;
                }
                double sigma = sigma1 + sigma2 + sigma3 + sigma4;
                if (sigma > bestSigma) {
                    bestSigma = sigma;
                    bestF[0] = f1;
                    bestF[1] = f2;
                    bestF[2] = f3;
                }
            }
        }
    }
}

unsigned char convert(unsigned char c, int f[3]) {
    if (c <= f[0]) {
        return 0;
    }
    if (c <= f[1]) {
        return 84;
    }
    if (c <= f[2]) {
        return 170;
    }
    return 255;
}

void convertImg(int f[3]) {
    if (threads >= 1) {
        omp_set_num_threads(threads);
    }
#pragma omp parallel for if (threads >= 0)
    for (int i = 0; i < img.size(); i++) {
        img[i] = convert(img[i], f);
    }
}


void readFile(string source) {
    ifstream in(source);

    if (!in) {
        throw invalid_argument("Failed to open file " + source);
    }

    in >> version;
```

```
    if (version != "P5") {
        throw invalid_argument("Invalid file version " + source);
    }

    if (!(in >> width >> height >> colorsNumber)) {
        throw invalid_argument("File is not supported: invalid data");
    }

    if (!in.get() || colorsNumber != 255) {
        throw invalid_argument("File is not supported: invalid data");
    }

    char color;

    while (in.get(color)) {
        img.push_back((unsigned char)(color));
    }

    if (img.size() != width * height) {
        throw invalid_argument("File is not supported: invalid data");
    }

    in.close();
}

void getHistogram() {
    for (int i = 0; i < img.size(); i++) {
        histogram[img[i]]++;
    }
}

void outFile(string source) {
    ofstream out(source);
    out << version << "\n";
    out << width << " " << height << "\n" << colorsNumber << "\n";
    for (int i = 0; i < img.size(); i++) {
        out << img[i];
    }
    if (out.bad()) {
        throw invalid_argument("Writing to " + source + " failed");
    }
    out.close();
}

int main(int argc, char* argv[]) {
    if (argc != 4) {
        cerr << "Incorrect number of arguments";
```

```
        return -1;
    }
    try {
        threads = atoi(argv[1]);
    }
    catch (exception& e) {
        cerr << "Incorrect number of threads";
        return -1;
    }
    try {
        readFile(argv[2]);
    }
    catch (invalid_argument& e) {
        cerr << e.what() << endl;
        return -1;
    }
    int f[3];
    double start = omp_get_wtime();
    getHistogram();
    recount();
    getBestF(f);
    convertImg(f);
    printf("%d %d %d\n", f[0], f[1], f[2]);
    printf("Time (%i thread(s)): %g ms\n", threads, (omp_get_wtime() - start)
* 1000);
    try {
        outFile(argv[3]);
    }
    catch (invalid_argument& e) {
        cerr << e.what() << endl;
        return -1;
    }
}
```