Лабораторная работа №4	М3139	2022
OpenMP	Матвеев Андрей Денисович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: Язык C++, компилятор Visual Studio 2019 x64

Описание

Необходимо написать программу, реализующую метод Оцу, использующую многопоточность, провести замеры времени при разных количествах потоков.

Вариант 2 – Метод Оцу

1. Описание конструкций OpenMP

OpenMP позволяет писать программы, использующие многопоточность. Для использования OpenMP необходимо импортировать библиотеку omp.h.

Любой блок кода, использующий OpenMP, должен быть обёрнут следующим образом:

```
#pragma omp parallel
{
    // код
}
```

У этого объявления есть один из параметров, который я использую – условное использование:

```
#pragma omp parallel if (/*логическое выражение*/)
```

Если логическое выражение верно, то код будет исполняться, используя многопоточность, иначе – код будет исполнятся на одном потоке без использования omp.

Чтобы распараллелить for блок, надо перед ним написать такой код:

```
#pragma omp for
```

У этого объявления есть параметр schedule, который в свою очередь имеет два параметра “type” и “chunk_size” (необязательный). type может быть либо static, либо dynamic (есть ещё варианты, но я ими не пользовался). Параметр chunk_size – это количество итераций for’а которое будет исполнено одним потоком. Если chunk_size * threads_num < n, где n – количество итераций в for’е, то каждому потоку достанется несколько “чанков” итераций, каждый из которых будет размером chunk_size. Если n не делится на chunk_size, то один из блоков будет меньше chunk_size.

Что значит static? Для краткости, обозначим chunk_size = x, тогда первому потоку достанутся итерации с номерами [0, x), второму потоку – [x, 2x), третьему – [3x, 4x) и т. д., если после этого остались итерации, то процесс заново запускается. Если аргумент chunk_size не указан, то он берётся таковым, чтобы на каждый поток досталось примерно одинаковое количество итераций. В случае dynamic, нет определённого порядка, в котором запускаются потоки. Если аргумент chunk_size не указан, то chunk_size = 1.

Последняя конструкция, которой я пользовался – atomic:

```
#pragma omp atomic
```

Эта конструкция заставляет следующую строчку кода исполняться атомарно, т. е. чтобы несколько потоков не выполняли эту строчку одновременно (например, инкремент переменной).

2. Описание работы написанного кода

Программа состоит из трёх файлов: otsu.h, otsu.cpp, hard.cpp. Файл otsu.cpp содержит реализацию алгоритма Оцу, otsu.h содержит объявления функций, структур и т. д., использующихся в otsu.cpp. hard.cpp содержит весь остальной код.

2.1 otsu.h

```
#pragma once

const int COLORS = 256;
const int LEVELS = 4;

struct Threshold {
    double sigma;
    int t0, t1, t2;

    Threshold() : sigma(0), t0(0), t1(0), t2(0) {}
    Threshold(double sigma, int t0, int t1, int t2) :
sigma(sigma), t0(t0), t1(t1), t2(t2) {}
};

void build_frequency_table(long long* frequency, char*
image, int size, int threads_count, int chunk_size = 1);
void build_table(double** main_table, long long* frequency,
int threads_count, int chunk_size = 1);
Threshold get_best_thresholds(double** main_table, int
threads_count, int chunk_size = 1);

    Threshold get_threshold(char* image, int size, int
    threads_count, int chunk_size = 1);
```

COLORS – количество уровней яркости в картинке. LEVELS – количество классов эквивалентности яркости, т. е. количество порогов + 1. Threshold – структура, позволяющая хранить пороги, и sigma – сигма, которая получается, при данных порогах. Остальное – функции, для работы алгоритма Оцу.

2.2 otsu.cpp

```
#include <cassert>
#include <iostream>
#include <omp.h>

#include "otsu.h"

void build_frequency_table(long long* frequency, char*
image, int size, int threads_count, int chunk_size) {
    for (int i = 0; i < COLORS; i++) {
        frequency[i] = 0;
    }
```

```
#pragma omp parallel if (threads_count != -1)
     {
#pragma omp for
          for (int i = 0; i < size; i++) {
               int normalized = image[i] < 0 ? image[i] + 256
: image[i];
#pragma omp atomic
               frequency[normalized]++;
          }
     }
}

void build_table(double** main_table, long long* frequency,
int threads_count, int chunk_size) {
#pragma omp parallel if (threads_count != -1)
     {
#pragma omp for
          for (int i = 1; i < COLORS; i++) {
               long long freq = 0;
               long long s = 0;
               for (int j = i; j < COLORS; j++) {
                    long long local_freq = frequency[j];
                    freq += frequency[j];
                    s += j * frequency[j];
                    if (freq != 0)
                         main_table[i][j] = (double)s * s /
freq;
               }
          }
     }
}

Threshold get_best_thresholds(double** main_table, int
threads_count, int chunk_size) {
     assert(LEVELS == 4);
     bool is_multithread = (threads_count != -1);
     threads_count = threads_count != -1 ? threads_count : 1;
     Threshold* local = new Threshold[threads_count];
#pragma omp parallel if (is_multithread)
     {
#pragma omp for
          for (int i = 1; i < COLORS; i++) {
               for (int j = i + 1; j < COLORS; j++) {
                    for (int k = j + 1; k < COLORS - 1; k++) {
```

```
                        double cur_sigma = main_table[1][i] +
main_table[i + 1][j] + main_table[j + 1][k] + main_table[k +
1][COLORS - 1];
                        Threshold& local_sigma =
local[omp_get_thread_num()];
                        if (local_sigma.sigma < cur_sigma) {
                             local_sigma =
Threshold(cur_sigma, i, j, k);
                        }
                   }
              }
         }
    }
    Threshold best_threshold;
    for (int i = 0; i < threads_count; i++) {
         if (best_threshold.sigma < local[i].sigma) {
              best_threshold = local[i];
         }
    }
    return best_threshold;
}

Threshold get_threshold(char* image, int size, int
threads_count, int chunk_size) {
    if (threads_count != -1)
         omp_set_num_threads(threads_count);
    long long* frequency = new long long[COLORS];
    double** main_table = new double* [COLORS];
    for (int i = 0; i < COLORS; i++) {
         main_table[i] = new double[COLORS];
    }
    build_frequency_table(frequency, image, size,
threads_count, chunk_size);
    build_table(main_table, frequency, threads_count,
chunk_size);
    Threshold res = get_best_thresholds(main_table,
threads_count, chunk_size);

    delete[] frequency;
    for (int i = 0; i < COLORS; i++) {
         delete[] main_table[i];
    }
    delete[] main_table;
```

```
    return res;
}
```

Главная функция, которая запускается из hard.cpp – это get_threshold. Она принимает картинку в виде массива char'ов (char* image) и её размер (int size). Также функция принимает параметры threads_count и chunk_size – количество потоков, которое будет использоваться в алгоритме, и chunk_size – соответствующий параметр для schedule. Сначала, мы создаём два массива frequency и main_table, а потом заполняем их соответствующими функциями: "build_frequency_table" и "build_table". Дальше запускаем функцию get_best_thresholds, которая на основании main_table ищет подходящие пороги и возвращает их вместе с сигмой. В конце функции освобождается память, которая была выделена под массивы.

Следующая функция, которую мы рассмотрим – build_frequency_table. Эта функция просто заполняет массив frequency, так, что в frequency[i] хранится количество появлений цвета i в картинке. Важное замечание, char – знаковый тип и хранит от -128 до 127, поэтому если какой-то цвет меньше нуля, мы прибавляем к нему 256. Здесь используется #pragma omp atomic, чтобы избежать состояния гонки:

```
#pragma omp atomic
frequency[normalized]++;
```

Дальше идёт функция build_table, которая заполняет двумерный массив main_table. Здесь надо сделать отступление и понять, как работает алгоритм Оцу. Основная его задача – найти пороги, тем самым разбив все пиксели на классы эквивалентности, так чтобы выделить цель на фоне. Пороги эти выбираются так, чтобы минимизировать взвешенную сумму внутрикластерных дисперсий для кластеров, сформированных после разделения по порогам. Это то же, что и установление порогов, обеспечивающих получение максимума межкластерной дисперсии.

Введём обозначения: $q_i = \sum_{f=f_i}^{f_{i+1}-1} p(f)$, где $p(f)$ – вероятность цвета f, f_i – значение i-го порога. $f_0 = 0$, $f_{m+1} = 256$, где m – количество порогов. $\mu_i = \sum_{f=f_i}^{f_{i+1}-1} \frac{f*p(f)}{q_i(f_i)}$ – среднее значение яркости i-го кластера, $\mu = \sum_{i=0}^{m} q_i \mu_i$ – среднее значение яркости в картинке. Наконец, межкластерная дисперсия равна сумме взвешенных квадратов расстояний между центрами кластеров и глобальным средним значением: $\sigma_B^2 = \sum_{i=0}^{m} q_i \, (\mu_i - \mu)^2 = \sum_{i=0}^{m} q_i \, {\mu_i}^2 - 2\mu^2 + \mu$. Так как последние два слагаемых – константы, то наша задача максимизировать Из этой формулы видно, что наша задача максимизировать $\sum_{i=0}^{m} q_i \, {\mu_i}^2$.

Вернёмся к функции build_table. Она заполняет массив main_table так, что $main\_table[i][j] = \sum_{f=i}^{j} q_f * \mu_f^2$. Вообще сумму на отрезке можно считать за O(255) используя префиксные суммы, но это всё равно не влияет на скорость работы. Можно заметить, вместо постоянного обращения к frequency[i], мы сохраняем это значение в локальную переменную local_freq = frequency[i], т. к. таким образом она сохранится где-то в регистрах и доступ к ней будет быстрее, чем обращение к оперативной памяти.

Теперь осталось рассмотреть функцию get_best_threshold. Она устроена очень просто, мы перебираем три порога и выбираем их так, чтобы σ_B^2 была максимальна. Чтобы избежать состояние гонки, создан массив local, где local[i] – максимальный threshold, среди тех, которые просмотрел i-ый поток, тогда ответ – это максимум среди всех local[i]. Также, чтобы посчитать $\sum_{i=0}^{m} q_i {\mu_i}^2$ используется такое выражение:

```
    double cur_sigma = (main_table[1][i] + main_table[i + 1][j]) + (main_table[j + 1][k] + main_table[k + 1][COLORS - 1]);
```

Первые два и последние два слагаемых взяты в скобки, потому что так процессор сможет их посчитать параллельно. Если бы скобок не было, то процессор бы считал их так: cur_sigma = ((s1 + s2) + s3) + s4, тратя 3 последовательных операции, вместо 2 параллельных + 1.

2.3 hard.cpp

```
#ifndef _CRT_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS
#endif // !_CRT_SECURE_NO_WARNINGS

#pragma GCC optimize("Ofast")
#pragma GCC optimize("O2");

#include <iostream>
#include <omp.h>
#include <fstream>
#include <string>
#include <cstring>
#include <stdexcept>

#include "otsu.h"

#define MAX_IMAGE_SIZE (1 << 30) // Exactly one gb
#define MAX_TREADS_COUNT 1000
```

```
using namespace std;

ifstream open_for_input(char* filename) {
     ifstream file;
     file.exceptions(file.exceptions() | ios::failbit);
     file.open(filename, ios::binary);
     return file;
}

ofstream open_for_output(char* filename) {
     ofstream file;
     file.exceptions(file.exceptions() | ios::failbit);
     file.open(filename);
     return file;
}

void print_file_error_message_and_exit() {
     if (ios::eofbit) {
          cerr << "End-Of-File reached while performing an
extracting operation on an input stream.\n";
     }
     else if (ios::failbit) {
          cerr << "The last input operation failed because of
an error related to the internal logic of the operation
itself.\n";
     }
     else if (ios::badbit) {
          cerr << "Error due to the failure of an
input/output operation on the stream buffer.\n";
     }
     else {
          cerr << "Unknown exception\n";
     }
     exit(1);
}

int main(int argc, char** argv)
{
     if (argc < 4) {
          std::cerr << "Must be at least 4 arguments, but
only " << argc << " were received\n";
          exit(1);
     }
```

```
    int threads_count = atoi(argv[1]);
    if (threads_count < -1 || threads_count >
MAX_TREADS_COUNT) {
        cerr << "Number of threads must lie in range [1,
1000] or [-1, -1]\n";
        exit(1);
    }

    ifstream input_file;
    ofstream output_file;
    try {
        input_file = open_for_input(argv[2]);
        output_file = open_for_output(argv[3]);
    }
    catch (ios_base::failure e) {
        cerr << e.what() << "\n" << strerror(errno) <<
"\n";
        exit(1);
    }

    char* image = nullptr;
    unsigned int width, height, max_color;
    string type;

    try {
        input_file >> type >> width >> height >> max_color;
        if (type != "P5" || max_color != 255 || width == 0
|| height == 0 || width * 1ll * height > MAX_IMAGE_SIZE) {
            throw invalid_argument("wrong file format");
        }
        image = new char[width * height];
        input_file.read(image, 1); // skipping \n
        input_file.read(image, width * 1ll * height);
    }
    catch (ios_base::failure e) {
        cerr << e.what() << "\n";
        print_file_error_message_and_exit();
    }
    catch (invalid_argument e) {
        cerr << e.what() << "\n";
        exit(1);
    }

    Threshold threshold;
```

```
    int epochs = 50;
    double start_time = omp_get_wtime();
    for (int i = 0; i < epochs; i++) {
        threshold = get_threshold(image, width * height,
threads_count);
    }
    double end_time = omp_get_wtime();
    double delta = end_time - start_time;
    printf("Time(% i thread(s)) : % g ms\n", threads_count,
delta * 1000 / epochs);
    printf("%u %u %u\n", threshold.t0, threshold.t1,
threshold.t2);

    try {
        output_file.write("P5\n", 3);
        output_file.write(to_string(width).c_str(),
to_string(width).size());
        output_file.write(" ", 1);
        output_file.write(to_string(height).c_str(),
to_string(height).size());
        output_file.write("\n255\n", 5);
        for (int i = 0; i < width * height; i++) {
            int pixel = image[i];
            if (pixel <= threshold.t0)
                pixel = 0;
            else if (pixel <= threshold.t1)
                pixel = 84;
            else if (pixel <= threshold.t2)
                pixel = 170;
            else
                pixel = 255;
            char temp[1];
            temp[0] = pixel >= 128 ? pixel - 256 : pixel;
            output_file.write(temp, 1);
        }

        input_file.close();
        output_file.close();
    }
    catch (ios_base::failure e) {
        cerr << e.what() << "\n";
        print_file_error_message_and_exit();
    }
```

```
    delete[] image;

    return 0;
}
```

Функция open_for_input() открывает файл для чтения в бинарном режиме, попутно заставляя этот файловый объект кидать исключения. Функция open_for_output() делает то же самое, но для записи в файл. Фукнция print_file_error_message_and_exit() выводит человекочитаемое сообщение об ошибке, связанной с файлом, и завершает программу. Теперь посмотрим на main.cpp: сначала открываются файлы, попутно проверяя аргументы командной строки на валидность, объявля.тся переменные. Дальше файл считывается в массив char* image, точнее только данные, относящиеся к самой картинке, а ширина, длина и остальные служебные данные записываются в соответствующие переменные.

Потом запускается сам алгоритм Оцу, а результат сохраняется в переменную Threshold threshold. Дальше уже идёт запись преобразованного изображения в выходной файл.

3. Лог вывода программы

Time(20 thread(s)) : 11.0708 ms

77 130 187

4. Графики

4.1 Разное количество потоков при schedule(static), без chunk_size (рисунок 1)

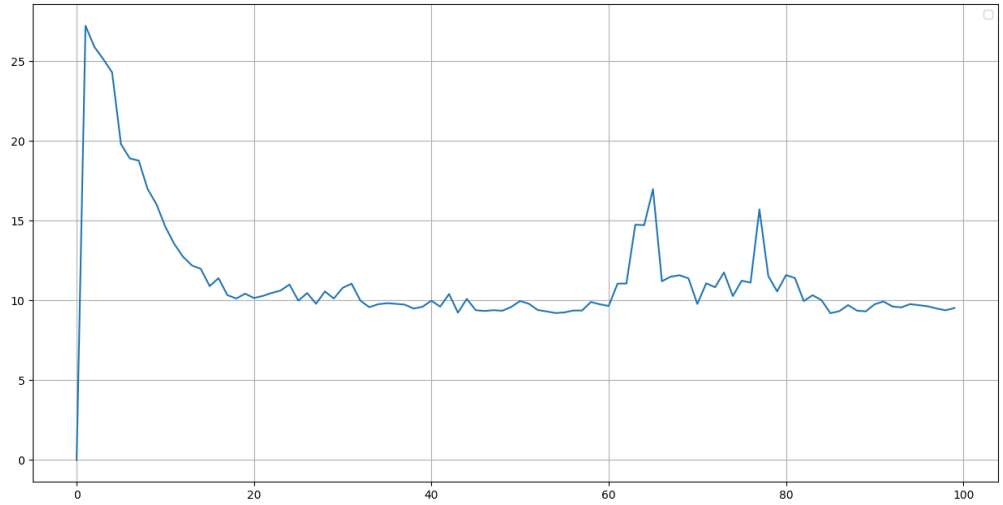

Рисунок 1 – график зависимости времени работы программы от количества потоков, при schedule(static)

По оси X – количество потоков, по оси Y – время работы программы в миллисекундах. Видно, что начиная с 20-ти потоков достигается минимум, а потом почти не уменьшается, потому что тратятся ресурсы на синхронизацию потоков, и из-за этого нету выигрыша во времени.

4.2 Разное количество потоков при schedule(dynamic), без chunk_size (рисунок 2)

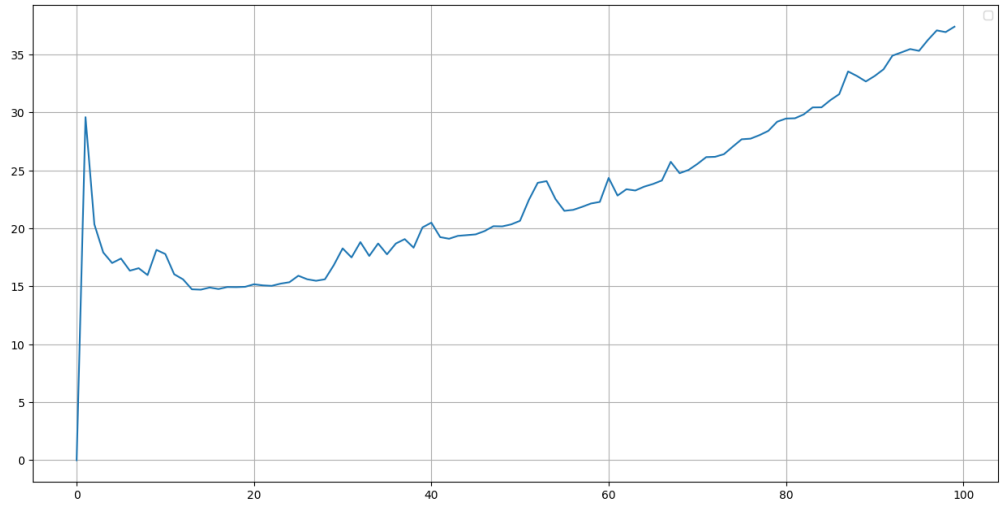

Рисунок 2 - график зависимости времени работы программы от количества потоков, при schedule(dynamic)

По оси X – количество потоков, по оси Y – время работы программы в миллисекундах. Опять, оптимальное значение достигается при 20-ти потоках, но потом время возрастает. Это очевидно, т. к. синхронизировать потоки при schedule(dynamic) сложнее (они идут в произвольном порядке). На это тратится очень много ресурсов.

4.3 Одинаковое количество потоков (20), но различный параметр chunk_size, с schedule(static) (рисунок 3)

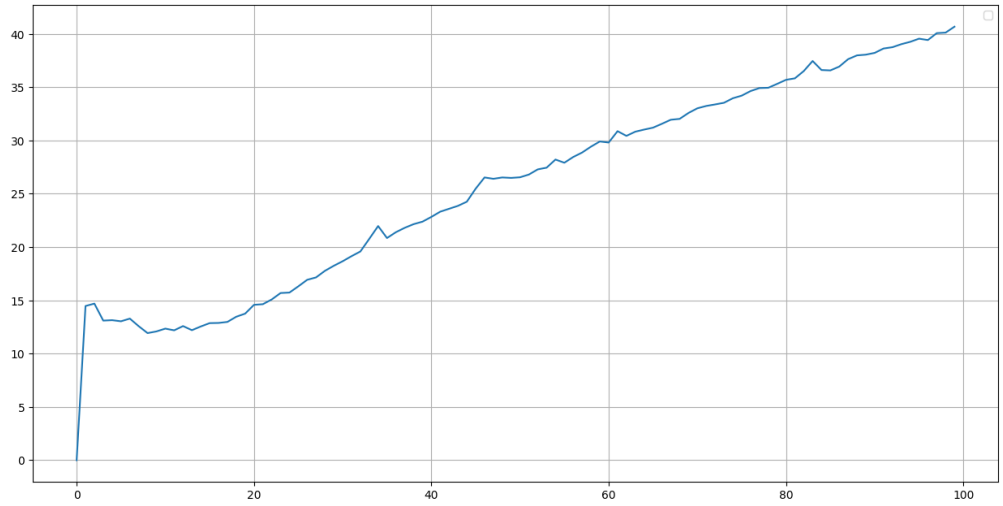

Рисунок 3 - график зависимости времени работы программы от chunk_size, при schedule(static)

По оси X – chunk_size, по оси Y – время работы программы в миллисекундах. Оптимальное значение при chunk_size = 10. Это связано с тем, что на ядре потоки работают поочерёдно (скажем для упрощения, что каждый поток работает время t) и при chunk_size = 10, наш поток как раз за время “t” обрабатывает все 10 итераций for’a.

4.4 Одинаковое количество потоков (20), но различный параметр chunk_size, с schedule(dynamic) (рисунок 4)

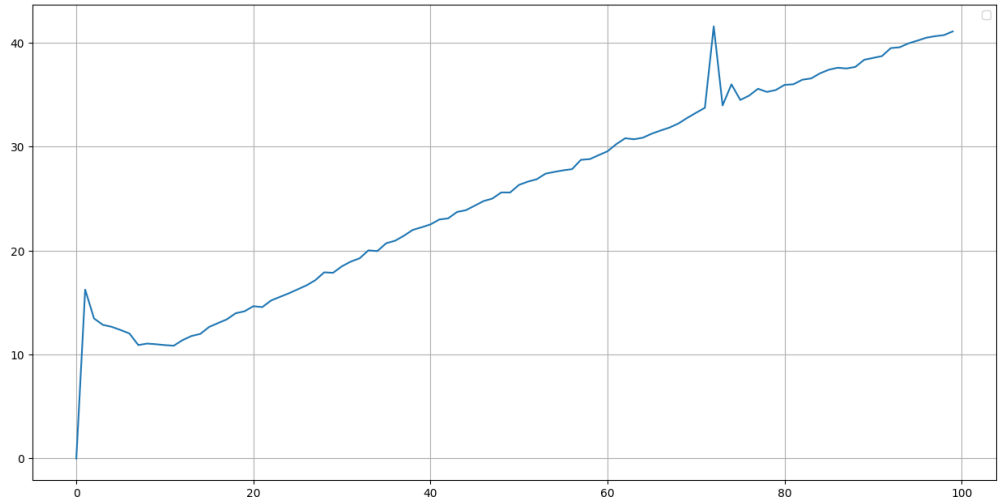

Рисунок 4 - график зависимости времени работы программы от chunk_size, при schedule(dynamic)

По оси X – chunk_size, по оси Y – время работы программы в миллисекундах. Здесь ситуация аналогична предыдущему графику, но есть резкий подъём на ~72-ом потоке. Скорее всего на ядре, где расположен этот поток, сидит ОС, или какая-нибудь другая ресурсоемкая вещь.

4.5 Выключенный openmp и включённый с 1 потоком, без schedule.

Здесь нет смысла приводить график, поэтому просто посмотрим результат:

Выключенный openmp : 25.5272 ms

Включенный, с одним потоком : 25.0612 ms

5. Источники

https://learn.microsoft.com/ru-ru/cpp/parallel/openmp/d-using-the-schedule-clause?view=msvc-170

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%9E%D1%86%D1%83

6. Листинг кода

hard.cpp

```
#ifndef _CRT_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS
#endif // !_CRT_SECURE_NO_WARNINGS

#pragma GCC optimize("Ofast")
#pragma GCC optimize("O2");

#include <iostream>
#include <omp.h>
#include <fstream>
#include <string>
#include <cstring>
#include <stdexcept>

#include "otsu.h"

#define MAX_IMAGE_SIZE (1 << 30) // Exactly one gb
#define MAX_TREADS_COUNT 1000

using namespace std;

ifstream open_for_input(char* filename) {
     ifstream file;
     file.exceptions(file.exceptions() | ios::failbit);
     file.open(filename, ios::binary);
     return file;
}

ofstream open_for_output(char* filename) {
     ofstream file;
     file.exceptions(file.exceptions() | ios::failbit);
     file.open(filename);
     return file;
}

void print_file_error_message_and_exit() {
     if (ios::eofbit) {
          cerr << "End-Of-File reached while performing an
extracting operation on an input stream.\n";
     }
     else if (ios::failbit) {
```

```
            cerr << "The last input operation failed because of
an error related to the internal logic of the operation
itself.\n";
        }
        else if (ios::badbit) {
            cerr << "Error due to the failure of an
input/output operation on the stream buffer.\n";
        }
        else {
            cerr << "Unknown exception\n";
        }
        exit(1);
}

int main(int argc, char** argv)
{
        if (argc < 4) {
            std::cerr << "Must be at least 4 arguments, but
only " << argc << " were received\n";
            exit(1);
        }

        int threads_count = atoi(argv[1]);
        if (threads_count < -1 || threads_count >
MAX_TREADS_COUNT) {
            cerr << "Number of threads must lie in range [1,
1000] or [-1, -1]\n";
            exit(1);
        }

        ifstream input_file;
        ofstream output_file;
        try {
            input_file = open_for_input(argv[2]);
            output_file = open_for_output(argv[3]);
        }
        catch (ios_base::failure e) {
            cerr << e.what() << "\n" << strerror(errno) <<
"\n";
            exit(1);
        }

        char* image = nullptr;
        unsigned int width, height, max_color;
```

```
    string type;

    try {
        input_file >> type >> width >> height >> max_color;
        if (type != "P5" || max_color != 255 || width == 0
|| height == 0 || width * 1ll * height > MAX_IMAGE_SIZE) {
            throw invalid_argument("wrong file format");
        }
        image = new char[width * height];
        input_file.read(image, 1); // skipping \n
        input_file.read(image, width * 1ll * height);
    }
    catch (ios_base::failure e) {
        cerr << e.what() << "\n";
        print_file_error_message_and_exit();
    }
    catch (invalid_argument e) {
        cerr << e.what() << "\n";
        exit(1);
    }

    Threshold threshold;

    int epochs = 50;
    double start_time = omp_get_wtime();
    for (int i = 0; i < epochs; i++) {
        threshold = get_threshold(image, width * height,
threads_count);
    }
    double end_time = omp_get_wtime();
    double delta = end_time - start_time;
    printf("Time(% i thread(s)) : % g ms\n", threads_count,
delta * 1000 / epochs);
    printf("%u %u %u\n", threshold.t0, threshold.t1,
threshold.t2);

    try {
        output_file.write("P5\n", 3);
        output_file.write(to_string(width).c_str(),
to_string(width).size());
        output_file.write(" ", 1);
        output_file.write(to_string(height).c_str(),
to_string(height).size());
        output_file.write("\n255\n", 5);
```

```
            for (int i = 0; i < width * height; i++) {
                int pixel = image[i];
                if (pixel <= threshold.t0)
                    pixel = 0;
                else if (pixel <= threshold.t1)
                    pixel = 84;
                else if (pixel <= threshold.t2)
                    pixel = 170;
                else
                    pixel = 255;
                char temp[1];
                temp[0] = pixel >= 128 ? pixel - 256 : pixel;
                output_file.write(temp, 1);
            }

            input_file.close();
            output_file.close();
        }
        catch (ios_base::failure e) {
            cerr << e.what() << "\n";
            print_file_error_message_and_exit();
        }

        delete[] image;

        return 0;
}
```

otsu.cpp

```
#include <cassert>
#include <iostream>
#include <omp.h>

#include "otsu.h"

void build_frequency_table(long long* frequency, char*
image, int size, int threads_count, int chunk_size) {
    for (int i = 0; i < COLORS; i++) {
        frequency[i] = 0;
    }
#pragma omp parallel if (threads_count != -1)
    {
#pragma omp for
```

```
          for (int i = 0; i < size; i++) {
               int normalized = image[i] < 0 ? image[i] + 256
: image[i];
#pragma omp atomic
               frequency[normalized]++;
          }
     }
}

void build_table(double** main_table, long long* frequency,
int threads_count, int chunk_size) {
#pragma omp parallel if (threads_count != -1)
     {
#pragma omp for
          for (int i = 1; i < COLORS; i++) {
               long long freq = 0;
               long long s = 0;
               for (int j = i; j < COLORS; j++) {
                    long long local_freq = frequency[j];
                    freq += frequency[j];
                    s += j * frequency[j];
                    if (freq != 0)
                         main_table[i][j] = (double)s * s /
freq;
               }
          }
     }
}

Threshold get_best_thresholds(double** main_table, int
threads_count, int chunk_size) {
     assert(LEVELS == 4);
     bool is_multithread = (threads_count != -1);
     threads_count = threads_count != -1 ? threads_count : 1;
     Threshold* local = new Threshold[threads_count];
#pragma omp parallel if (is_multithread)
     {
#pragma omp for
          for (int i = 1; i < COLORS; i++) {
               for (int j = i + 1; j < COLORS; j++) {
                    for (int k = j + 1; k < COLORS - 1; k++) {
                         double cur_sigma = main_table[1][i] +
main_table[i + 1][j] + main_table[j + 1][k] + main_table[k +
1][COLORS - 1];
```

```
                        Threshold& local_sigma =
local[omp_get_thread_num()];
                        if (local_sigma.sigma < cur_sigma) {
                             local_sigma =
Threshold(cur_sigma, i, j, k);
                        }
                   }
              }
         }
    }
    Threshold best_threshold;
    for (int i = 0; i < threads_count; i++) {
         if (best_threshold.sigma < local[i].sigma) {
              best_threshold = local[i];
         }
    }
    return best_threshold;
}

Threshold get_threshold(char* image, int size, int
threads_count, int chunk_size) {
    if (threads_count != -1)
         omp_set_num_threads(threads_count);
    long long* frequency = new long long[COLORS];
    double** main_table = new double* [COLORS];
    for (int i = 0; i < COLORS; i++) {
         main_table[i] = new double[COLORS];
    }
    build_frequency_table(frequency, image, size,
threads_count, chunk_size);
    build_table(main_table, frequency, threads_count,
chunk_size);
    Threshold res = get_best_thresholds(main_table,
threads_count, chunk_size);

    delete[] frequency;
    for (int i = 0; i < COLORS; i++) {
         delete[] main_table[i];
    }
    delete[] main_table;

    return res;
   }
```

otsu.h

```
#pragma once

const int COLORS = 256;
const int LEVELS = 4;

struct Threshold {
     double sigma;
     int t0, t1, t2;

     Threshold() : sigma(0), t0(0), t1(0), t2(0) {}
     Threshold(double sigma, int t0, int t1, int t2) :
sigma(sigma), t0(t0), t1(t1), t2(t2) {}
};

void build_frequency_table(long long* frequency, char*
image, int size, int threads_count, int chunk_size = 1);
void build_table(double** main_table, long long* frequency,
int threads_count, int chunk_size = 1);
Threshold get_best_thresholds(double** main_table, int
threads_count, int chunk_size = 1);
    Threshold get_threshold(char* image, int size, int
threads_count, int chunk_size = 1);
```