Лабораторная работа №4	Б10	2022
OpenMP	Хорохорин Андрей Сергеевич	

Цель работы

Знакомство с основами многопоточного программирования.

Инструментарий

Секция генерируется автоматически при компиляции

1. clang version 14.0.6
2. VIM - Vi IMproved 9.0 (2022 Jun 28, compiled Nov 19 2022 14:37:14)
3. GNU Make 4.3
4. XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022/Arch Linux)
5. OpenMP standard 2.0

Описание конструкция OpenMP для распараллеливания команд

В рамках работы мною были использованы следующие директивы OpenMP:

- `#pragma omp parallel`
  Объявляет начало блока, в котором код будет исполняться параллельно. Весь код вне этой директивы исполняется одним главным потоком.
- `#pragma omp for schedule(type, chunk)`
  Явно задаёт способ выполнения цикла на нескольких потоках. А именно каждая итерация объявляется как задача, а затем эти задачи распределяются по потокам. Само распределение может быть реализовано различным образом: за это отвечают аргументы type и chunk. Более подробно про различные распределения и их эффективность в данной задаче можно ознакомиться в секции «Экспериментальная часть».
- `#pragma omp critical`
  Объявляет некоторый блок кода как критическую секцию, также называемую mutex. Данная секция в каждый момент времени может исполняться лишь одним потоком.
- `#pragma omp threadprivate(variables)` По умолчанию все переменные объявленный внутри `omp parallel` являются локальными для потока, а все объявленные вне её: за глобальные. Данная директива позволяет сделать исключение для некоторого списка переменных, которые будут объявлены вне `omp parallel`, но будут созданными для каждого из потоков в отдельности.

Помимо директив используются следующие runtime функции библиотеки:

- `omp_get_wtime()`
  Возвращает астрономическое время, прошедшее после запуска программы.
- `omp_set_num_threads(n)`
  Устанавливает количество потоков по умолчанию, которые будут использованы для параллельного исполнения параллельных секций OpenMP.

Описание работы написанного кода

Глобально можно поделить программу на 2 логические части. Первая из них это работа с изображением, включающее его ввод/вывод и трансформацию, принимающую функцию, которая будет применена ко всем пикселям изображения. Всё перечисленное реализовано в классе `PgmImage`. Думаю, не стоит отдельно останавливаться на реализации данного класса, так как ничего интересного в ней нет.
Вторая часть это сам алгоритм Оцу, определяющий по гистограмме цветов пороговые значения. Он реализован дважды: с директивами openmp и без.
Ps. Можно было конечно завернуть все обращения к функциям из openmp в ifdef, компилировать в make две версии программы: с fopenmp и без, а затем запускать нужную из них(в случае -1 в качестве первого аргумента бинарник без openmp, иначе с ним) при помощи bash скрипта. Но перелопачивать github CI мне не хочется.
Общая идея алгоритма следующая: изначально насчитываются взвешенные и не взвешенные префиксные суммы для гистограммы цветов: `hist_prefix` и `w_hist_prefix` соответственно. Далее рекурсивной процедурой `get_partition` перебираются все разбиения и в функции `relax_ans` для текущего разбиения подсчитывается целевая функция и, если смогли улучшить свой прошлый ответ, то обновляем его на текущий.

Предподсчёт

Функция `OtsuMultithread` является точкой входа в реализацию алгоритма. В ней насчитывается `hist_prefix` и `w_hist_prefix`, начинается параллельная секция и инициализируются массивы `cur_part` для каждого из потоков, так как благодаря директиве `#pragma omp threadprivate(cur_part)` он является локальным для каждого из потоков. Это необходимо, так как каждый из потоков перебирает своё подмножество разбиений и очевидно, что для хранения текущего разбиения каждого потока должна быть своя память.
После описанных выше действий запускается функция `get_partition`, осуществляющая перебор разбиений.

```
#pragma omp threadprivate(cur_part)
vector<uint8_t> OtsuMultithread(const vector<unsigned int>& hist) {
    N = std::accumulate(hist.begin(), hist.end(), 0);
    hist_prefix.resize(MAXP + 1, 0);
    w_hist_prefix.resize(MAXP + 1, 0);

    for (size_t i = 1; i <= MAXP; ++i) {
```

```
        hist_prefix[i] = hist_prefix[i - 1] + hist[i - 1];
        w_hist_prefix[i] = w_hist_prefix[i - 1] + 1ll * hist[i - 1] * (i - 1);
    }

#pragma omp parallel
    {
        cur_part.resize(M - 1);
        get_partition(1, 0);
    }
    return part;
}
```

Перебор разбиений

Реализация перебора является рекурсивной, благодаря чему работает для произвольного количества цветовых зон, на которые делится изображение.
Важной деталью реализации тут является то, что распараллеливается только цикл перебирающий первую границу разбиения. В противном случае слишком много маленьких итераций, что работает на порядок медленнее, чем данное решение.
Отметим то, что итерации в моей реализации имеют различную вычислительную сложность, так как чем меньше первая граница, тем больше существует разбиений оставшейся части цветов. Эта деталь во многом определяет выбор оптимального параметра для `schedule`.

```
void get_partition(int tl, int cur_th) {
    if (cur_th == M - 1) {
        relax_ans();
    } else if (cur_th == 0) {
#pragma omp for schedule(dynamic, 1)
        for (size_t i = tl; i < MAXP; ++i) {
            cur_part[cur_th] = i;
            get_partition(i + 1, cur_th + 1);
        }
    } else {
        for (size_t i = tl; i < MAXP; ++i) {
            cur_part[cur_th] = i;
            get_partition(i + 1, cur_th + 1);
        }
    }
}
```

Обновление ответа

В функции `relax_ans`, насчитывается значение целевой функции для текущего разбиения на слагаемые, пользуясь подсчитанными `hist_prefix` и `w_hist_prefix`. Отдельно стоит отметить, что в данной функции есть запись в общие для потоков ячейки памяти `part` и `part_disp` и если два потока захотят одновременно обновить в них значения получится не пойми что, поэтому необходимо сделать обновление ответа в критической секцией.
Помимо этого, просто глядя на код не понятно, почему на строчках 19-27 написан такой велосипед. Почему же нельзя было просто обойтись одним if внутри критической

секции? Потому что в таком случае это условие являлось бы бутылочным горлышком нашего алгоритма: потоки постоянно ждали бы доступа к проверке того, что их текущий ответ хуже, чем уже найденный, что происходит почти всегда. Такая реализация будет работать гораздо хуже даже однопоточной.

```
// [tl, tr)
std::pair<unsigned long long, double> get_sums_for_part(int tl, int tr) {
    unsigned long long q = get_sum(hist_prefix, tl, tr);
    double mu = 1.0 * get_sum(w_hist_prefix, tl, tr) / q;
    return {q, mu};
}

void relax_ans() {
    double cur_disp = 0;
    auto [q, mu] = get_sums_for_part(0, cur_part.front());
    cur_disp += q * mu * mu;
    for (size_t i = 1; i < cur_part.size(); ++i) {
        std::tie(q, mu) = get_sums_for_part(cur_part[i - 1], cur_part[i]);
        cur_disp += q * mu * mu;
    }
    std::tie(q, mu) = get_sums_for_part(cur_part.back(), MAXP);
    cur_disp += q * mu * mu;

    if (cur_disp > part_disp) {
#pragma omp critical
        {
            if (cur_disp > part_disp) {
                part = cur_part;
                part_disp = cur_disp;
            }
        }
    }
}
```

Результат работы программы

Секция генерируется автоматически при компиляции
Модель процессора, на котором проводилось тестирование:

```
model name : AMD A12-9720P RADEON R7, 12 COMPUTE CORES 4C+8G
Time (-1 thread(s)): 0.0892451 ms 77 130 187
```

Экспериментальная часть

Список источников

- Документация по OpenMP

Листинг кода

Секция генерируется автоматически при компиляции

Listing 1: **../include/OtsuMultithread.h**

```
#ifndef OTSU_MULTITHREAD_GUARD
#define OTSU_MULTITHREAD_GUARD

#include <cstdint>
#include <vector>

std::vector<uint8_t> OtsuMultithread(const std::vector<unsigned int>& hist);

#endif
```

Listing 2: **../include/OtsuSinglethread.h**

```
#ifndef OTSU_SINGLETHREAD_GUARD
#define OTSU_SINGLETHREAD_GUARD

#include <vector>
#include <inttypes.h>

using std::size_t;
using std::vector;
```

```
std::vector<uint8_t> OtsuSinglethread(const std::vector<unsigned int>& hist);

#endif
```

Listing 3: **../include/PgmImage.h**

```
#ifndef PGM_IMAGE_GUARD
#define PGM_IMAGE_GUARD

#include <string>
#include <vector>
#include <cassert>
#include <iostream>
#include <functional>

struct PgmImage {
    const std::string magic_bits = "P5";

    unsigned int height, width;
    std::vector<unsigned int> hist;
    std::vector<std::vector<uint8_t>> pixels;

    PgmImage(std::istream& is);

    void transform(std::function<uint8_t(uint8_t)> transformer);

};

std::ostream& operator << (std::ostream& os, const PgmImage& p);


#endif
```

Listing 4: **../src/hard.cpp**

```
#include <cstdint>
#include <omp.h>

#include <iostream>
#include <fstream>
#include <utility>
#include <vector>
#include <cassert>
#include <tuple>
#include <algorithm>
#include <numeric>
#include <functional>

#include "OtsuSinglethread.h"
#include "OtsuMultithread.h"
#include "PgmImage.h"

const std::vector<int> gray_scale = {0, 84, 170, 255};

int main(int argc, char* argv[]) {
    if (argc != 4) {
        std::cout << "Synopsis: ./omp4 <treads> <input file> <output file>\n";
        return 1;
```

```
    }

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

    std::ifstream fin(argv[2], std::ios::binary);
    if (!fin) {
        std::cout << "Can not read from file with name " << argv[2] << '\n';
        return 1;
    }
    PgmImage picture(fin);

    std::vector<uint8_t> partition;
    double start_time = omp_get_wtime();
    if (threads == -1) {
        partition = OtsuSinglethread(picture.hist);
    } else {
        if (threads != 0) omp_set_num_threads(threads);
        partition = OtsuMultithread(picture.hist);
    }
    std::cout <<  "Time (" << threads << " thread(s)): " << \
        omp_get_wtime() - start_time << " ms\n";

    std::function<uint8_t(uint8_t)> trans = [&partition](uint8_t x) {
        uint8_t gray_lvl = std::find_if(partition.begin(), partition.end(),
                [&x] (uint8_t threshold) {
                return x < threshold;
            }
        ) - partition.begin();
        return gray_scale[gray_lvl];
    };
    picture.transform(trans);

    std::ofstream fout(argv[3], std::ios::binary);
    if (!fout) {
        std::cout << "Can not write to file with name" << argv[3] << '\n';
        return 1;
    }
    fout << picture;

    for (uint8_t i : partition) std::cout << i - 1 << ' ';
    std::cout << '\n';

    return 0;
}
```

Listing 5: **../src/OtsuMultithread.cpp**

```
#include "OtsuMultithread.h"

#include <vector>
#include <tuple>
#include <algorithm>
#include <numeric>

using std::vector;

namespace {

const int M = 4;
const size_t MAXP = UINT8_MAX + 1;
vector<uint8_t> part;
```

```
unsigned int N = 0;
double part_disp = 0;
vector<uint8_t> cur_part;
vector<unsigned long long> hist_prefix, w_hist_prefix;

#pragma omp threadprivate(cur_part)

// [tl, tr)
unsigned long long get_sum(const vector<unsigned long long>& pref_sums,
        int tl, int tr) {
    return pref_sums[tr] - pref_sums[tl];
}

// [tl, tr)
std::pair<unsigned long long, double> get_sums_for_part(int tl, int tr) {
    unsigned long long q = get_sum(hist_prefix, tl, tr);
    double mu = 1.0 * get_sum(w_hist_prefix, tl, tr) / q;
    return {q, mu};
}

void relax_ans() {
    double cur_disp = 0;
    auto [q, mu] = get_sums_for_part(0, cur_part.front());
    cur_disp += q * mu * mu;
    for (size_t i = 1; i < cur_part.size(); ++i) {
        std::tie(q, mu) = get_sums_for_part(cur_part[i - 1], cur_part[i]);
        cur_disp += q * mu * mu;
    }
    std::tie(q, mu) = get_sums_for_part(cur_part.back(), MAXP);
    cur_disp += q * mu * mu;

    if (cur_disp > part_disp) {
#pragma omp critical
        {
            if (cur_disp > part_disp) {
                part = cur_part;
                part_disp = cur_disp;
            }
        }
    }
}

void get_partition(int tl, int cur_th) {
    if (cur_th == M - 1) {
        relax_ans();
    } else if (cur_th == 0) {
#pragma omp for schedule(dynamic, 1)
        for (size_t i = tl; i < MAXP; ++i) {
            cur_part[cur_th] = i;
            get_partition(i + 1, cur_th + 1);
        }
    } else {
        for (size_t i = tl; i < MAXP; ++i) {
            cur_part[cur_th] = i;
            get_partition(i + 1, cur_th + 1);
        }
    }
}

} //end anonymous namespace
```

```
vector<uint8_t> OtsuMultithread(const vector<unsigned int>& hist) {
    N = std::accumulate(hist.begin(), hist.end(), 0);
    hist_prefix.resize(MAXP + 1, 0);
    w_hist_prefix.resize(MAXP + 1, 0);

    for (size_t i = 1; i <= MAXP; ++i) {
        hist_prefix[i] = hist_prefix[i - 1] + hist[i - 1];
        w_hist_prefix[i] = w_hist_prefix[i - 1] + 1ll * hist[i - 1] * (i - 1);
    }

#pragma omp parallel
    {
        cur_part.resize(M - 1);
        get_partition(1, 0);
    }
    return part;
}
```

Listing 6: **../src/OtsuSinglethread.cpp**

```
#include "OtsuSinglethread.h"

#include <vector>
#include <tuple>
#include <algorithm>
#include <numeric>

using std::vector;

namespace {

const int M = 4;
const size_t MAXP = UINT8_MAX + 1;
vector<uint8_t> part;

unsigned int N = 0;
double part_disp = 0;
vector<uint8_t> cur_part;
vector<unsigned long long> hist_prefix, w_hist_prefix;


// [tl, tr)
unsigned long long get_sum(const vector<unsigned long long>& pref_sums,
        int tl, int tr) {
    return pref_sums[tr] - pref_sums[tl];
}

// [tl, tr)
std::pair<unsigned long long, double> get_sums_for_part(int tl, int tr) {
    unsigned long long q = get_sum(hist_prefix, tl, tr);
    double mu = 1.0 * get_sum(w_hist_prefix, tl, tr) / q;
    return {q, mu};
}

void relax_ans() {
    double cur_disp = 0;
    auto [q, mu] = get_sums_for_part(0, cur_part.front());
    cur_disp += q * mu * mu;
    for (size_t i = 1; i < cur_part.size(); ++i) {
        std::tie(q, mu) = get_sums_for_part(cur_part[i - 1], cur_part[i]);
        cur_disp += q * mu * mu;
```

```
    }
    std::tie(q, mu) = get_sums_for_part(cur_part.back(), MAXP);
    cur_disp += q * mu * mu;

    if (cur_disp > part_disp) {
        part = cur_part;
        part_disp = cur_disp;
    }
}

void get_partition(int tl, int cur_th) {
    if (cur_th == M - 1) {
        relax_ans();
    } else {
        for (size_t i = tl; i < MAXP; ++i) {
            cur_part[cur_th] = i;
            get_partition(i + 1, cur_th + 1);
        }
    }
}

} //end anonymous namespace


vector<uint8_t> OtsuSinglethread(const vector<unsigned int>& hist) {
    N = std::accumulate(hist.begin(), hist.end(), 0);
    hist_prefix.resize(MAXP + 1, 0);
    w_hist_prefix.resize(MAXP + 1, 0);

    for (size_t i = 1; i <= MAXP; ++i) {
        hist_prefix[i] = hist_prefix[i - 1] + hist[i - 1];
        w_hist_prefix[i] = w_hist_prefix[i - 1] + 1ll * hist[i - 1] * (i - 1);
    }

    cur_part.resize(M - 1);
    get_partition(1, 0);
    return part;
}
```

Listing 7: **../src/PgmImage.cpp**

```
#include "PgmImage.h"

#include <iostream>

std::ostream& operator << (std::ostream& os, const PgmImage& p) {
    os << p.magic_bits << '\n' << p.width << ' ' << p.height << "\n255\n";
    for (const auto& row : p.pixels) {
        for (const uint8_t& pixel : row) {
            os << pixel;
        }
    }
    return os;
}

PgmImage::PgmImage(std::istream& is): hist(UINT8_MAX + 1, 0) {
    std::string magic;
    is >> magic;
    assert(magic == magic_bits && "unexpected image file magic bits");

    is >> width;
```

```
    is >> height;
    pixels.resize(height, std::vector<uint8_t> (width));

    int max_color;
    is >> max_color;
    assert(max_color == 255 && "unsupported color-space");
    is.ignore(2, '\n');

    for (size_t x = 0; x < height; ++x) {
        for (size_t y = 0; y < width; ++y) {
            uint8_t c;
            is.read((char*)&c, 1);
            hist[c]++;
            pixels[x][y] = c;
        }
    }

    assert(!is.bad() && "an error occurred while reading the file");
}

void PgmImage::transform(std::function<uint8_t(uint8_t)> transformer) {
    for (auto& row : pixels) {
        for (uint8_t& pixel : row) {
            pixel = transformer(pixel);
        }
    }
}
```