Лабораторная работа №4	B10	2022
OpenMP	Малов Тимофей Иванович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: работа должна быть выполнена на C или C++. В отчёте указать язык и компилятор, на котором вы работали. Стандарт OpenMP 2.0.

Описание

Необходимо написать программу, решающую одну из задач, описанных в разделе Варианты. Чем сложнее вариант, тем больше баллов за работу вы можете получить.

Помимо написания программы, необходимо провести замеры времени работы на вашем компьютере и привести графики времени работы программы (некоторые графики из следующих подпунктов можно объединить в один):

1. при различных значениях числа потоков при одинаковом параметре schedule* (без chunk_size);
2. при одинаковом значении числа потоков при различных параметрах schedule* (с chunk_size);
3. с выключенным openmp и с включенным с 1 потоком.

* `schedule(kind[, chunk_size])` – kind принимает значение static или dynamic, chunk_size – 1 и несколько (1-2) других значений

В каждом варианте результат работы программы выводится в выходной файл, а время работы программы - в поток вывода (stdout). Формат вывода (в формате Си): `“Time (%i thread(s)): %g ms\n”`. Время

работы выводится только в консоль. В данном случае имеется в виду время работы алгоритма без времени на считывание данных и вывод результата.

Для минимизации влияния степени загруженности процессора другими процессами, время должно усредняться по нескольким запускам.

Время в программе нужно измерять при помощи `omp_get_wtime()` (раздел 3.3).

Про `schedule` также можно почитать в спецификации (разделы 2.4.1 и Appendix D).

Конструкции OpenMp

1) **parallel [num_thread(k)]:** создаёт k (или дефолтное число, если k не указано) вычислительных блоков, каждый из которых выполняет следующий блок {} или команду параллельно, после чего объединяя все вычислительные блоки (то есть программа дальше не выполняется, пока все блоки не завершат работу).
2) **for [schedule [(static[, k]) / (dynamic[, k])]] [collapse(f)]:** запускает параллельное вычисление следующей команды for, по разному в зависимости от schedule:
    a) schedule(static[, k]): если цикл выполняет t итераций, то i-ый из n вычислительных блоков выполняет итерации [i * t / n; (i + 1) * t / n]. Если же указано ещё и число k, то цикл делится на блоки по k, которые раздаются потокам по модулю.
    b) schedule(dynamic[, k]): итерации цикла исполняются по k итераций цикла любым свободным вычислительным блоком. Если k не указано, то оно считается за 1.

    Если указано collapse(f), то параллельно запускается внутренность цикла вложенности f. При этом области перебора в этих циклах должны быть независимы друг от друга, иначе поведение неспецифицированно.

3) **parallel for:** эквивалентно двум последовательным parallel и for.
4) **atomic [read / write]:** гарантирует, что команда присвоения или чтения на следующей строке выполняется одновременно только одним вычислительным блоком, гарантируя thread-safety. Можно специфицировать действие какого типа должно быть атомарным с помощью read или write/
5) **critical [(label)]:** следующий {} блок будет выполняться только одним вычислителем в каждый момент. Если же указан label, то в любой момент будет выполнятся лишь один блок с таким label во всех файлах программы.
6) **barrier:** эта инструкция пишется внутри блока parallel, гарантируя, что все вычислительные блоки дождутся выполнения друг друга в этот момент, приостанавливая свои вычисления, пока все другие их не закончат.

Описание работы кода

Программа состоит из нескольких блоков:

1) **Error handling**(utility.hpp), который используется во всей программе: обработка ошибок в моей программе осуществляется за счёт std::variant. Я написал несколько удобных классов и синонимов: error - структура, в которой хранится одна строка с ошибкой с единственным explicit конструктором. Синоним or_error обозначает либо корректный результат вычисления T, либо текстовую ошибку error. maybe_error нужна для void функций, которые могут завершится с ошибкой. Далее написаны несколько удобных функций для обработки ошибок. Также в файле объявлен удобный синоним pixel, означающий байт пикселя картинки.
2) **Ввод/вывод**(файлы noexcept_io.cpp/hpp): reader реализован совсем просто - в нём хранится поток ввода ifstream и весь считанный файл в формате vector<char>, который парсится соответствующими функциями. read_char пытается вернуть следующий байт файла, если он есть. read_int пытается считать следующее число, если оно есть. skip_spaces пропускает все пробельные символы с текущей позиции до следующего непробельного. Если что-то сделать невозможно в какой-либо из этих функций, то они вернут error с корректным текстом. writer устроен ещё проще - это просто поток ofstream с двумя функциями write и writeln, которые пишут в файл данные, сообщая об ошибке, если сделать этого не получилось.
3) **Парсинг**(pgm_parsing.cpp/hpp): parse_pgm просто последовательно считывает картинку из данного reader'a, возвращая ошибку, если очередной байт считать не удалось, возвращая на выходе картинку в формате vector<vector<pixel>>. write_pgm так же последовательно пишешь картинку в файл, возвращая ошибку, если записать не получилось.
4) **Непосредственно алгоритм**(otsu.cpp/otsu.hpp): для начала посмотрим на формулу, которую требуется посчитать для каждых значений порогов - это сумма по i от 0 до 4:

$$q_i * \mu_i^2 = q_i/q_i^2 * (\sum_{f=l_{i-1}+1}^{l_i} f * p(f))^2 =$$
$$(\sum_{f=l_{i-1}+1}^{l_i} f * n_f)^2/N^2/(\sum_{f=l_{i-1}+1}^{l_i} n_f/N) =$$
$$(\sum_{f=l_{i-1}+1}^{l_i} f * n_f)^2/(N * \sum_{f=l_{i-1}+1}^{l_i} n_f)$$

Так как / N есть в каждом слагаемом при всех порогах, то на минимизацию оно не влияет, поэтому его можно не делать. Также, вместо того, чтобы считать эти две суммы для каждых границ можно предподсчитать частотность пикселей в картинке и префиксные суммы для двух таких выражений до перебора порогов, тогда подсчёт дисперсии для фиксированных порогов будет происходить за константное число арифметических операций. Соответствующие префиксные суммы считаются в функции calc_pref_sums, значение суммы формул выше в функции get_disp_sum. Далее, для реализации алгоритма в одном потоке достаточно лишь посчитать перебрать все возможные возрастающие пороги и выбрать такие, для которых дисперсия максимальна - всё это и происходит в функции calc_max_levels_one_thread. В итоге задача свелась к поиску максимума функции на отрезке, параллелить такую задачу можно так: параллельно вычисляем максимум на подотрезках, после чего берём максимум среди них. Проблема возникает в том, что теперь, из-за ограничений omp(параллелить мы можем только цикл вложенности 1), придётся перебрать не три возрастающий границы, а число из трёх цифр в системе счисления 256, после чего проверять, что цифры - которым и равны текущие перебираемые границы - идут в возрастающем порядке. Это увеличивает число оборотов цикла, относительно однопоточного решения в $256^3/C_{256}^3 \simeq 6$ раз. Внутри каждого потока(внутри блока parallel) создаются локальные максимумы, после чего весь внутренний цикл каким-либо способом разбивается на потоки, после чего все локальные максимумы “сливаются” в один глобальный в critical блоке. Разница между calc_max_levels_default и calc_max_levels_multithread в том,

что в default цикл разбивается динамически с дефолтным числом потоков и принимает, на блоки какого размера надо бить цикл, а в multithread цикл разбивается статически и принимает на какое число потоков он должен разбиваться. Число блоков везде указано константой 256, так как анализ показал её, как самую оптимальную.

5) **Измерения**(measurements.cpp/hpp): все три функции для всех возможных конфигураций (measure_threads_num для разного числа потоков, measure_dynamic_blocks для разного размера этих блоков, а measure_one_thread всего единожды) 30 раз запускают алгоритм: считают оптимальные пороги и применяют их к картинке, считая суммарное время. После чего выводят информацию на экран.
6) **Точка старта**(hard.cpp): в main открываются reader и writer для данного файла, парсится картинка, и запускается код либо на измерения (measure_…), либо обычным запуском(normal_start). В normal_start в зависимости от числа требуемых ядер запускается нужная реализация и в правильном формате на экран выводится результат.

Результат работы программы

Процессор, на котором производилось тестирование: AMD Ryzen 5 4600H

```
Time (0 thread(s)): 240.412 ms
77 130 187
```

Экспериментальная часть

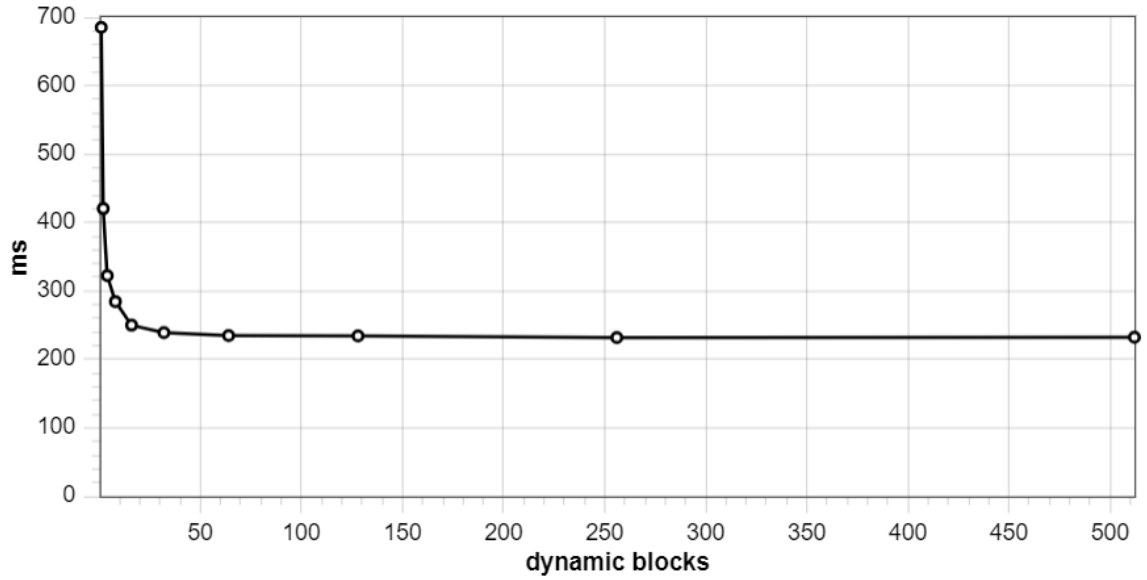

График 1 - зависимость времени работы от числа блоков в динамическом разбиении

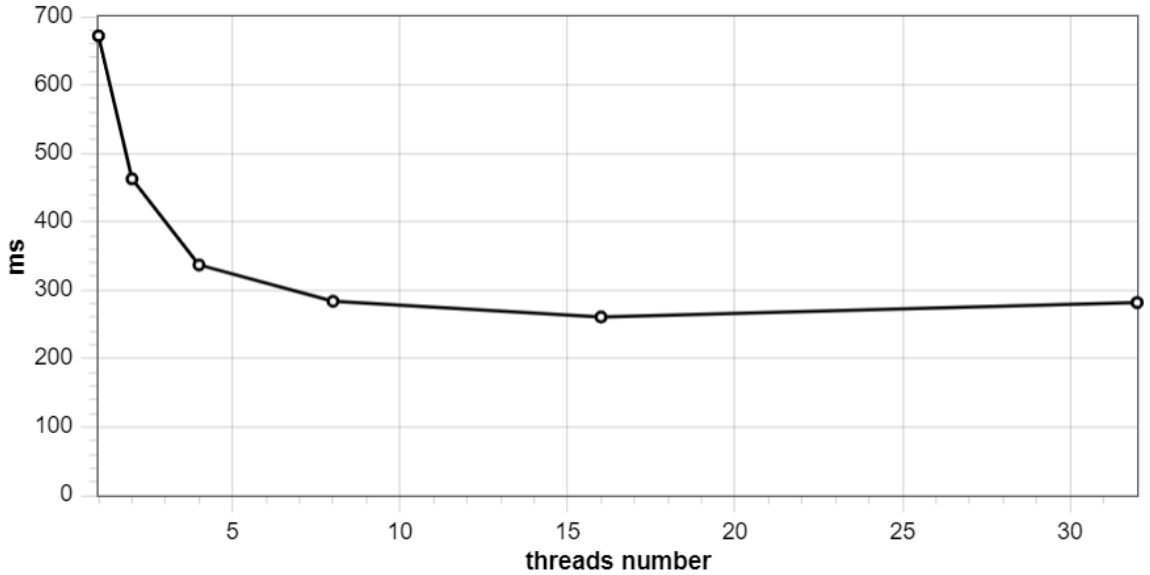

График 2 - зависимость времени работы от числа ядер со статическим разбиением

Список источников

1) https://drive.google.com/file/d/122YfkZEbzzHFtoSx-LwwUHpg7rmqiNVw/view - ваша ссылка с описанием алгоритма
2) https://netpbm.sourceforge.net/doc/pgm.html - описание формата картинки

3) https://smallpond.ca/jim/photomicrography/pgmViewer/index.html - отображалка картинок такого формата
4) https://www.graphreader.com/plotter - сайт, где были нарисованы графики

Листинг кода

hard.cpp:

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

#include "utility.hpp"
#include "otsu.hpp"
#include "measurements.hpp"
#include "pgm_parsing.hpp"

using std::cerr;
using std::cout;
using std::vector;
using std::string;
using std::to_string;
using namespace noexcept_io;

void normal_start(vector<vector<pixel>>& pic, int threads_num,
noexcept_writer& writer) {
	double start = omp_get_wtime();
	vector<int> max_levels;
	if (threads_num == -1)
		max_levels = calc_max_levels_one_thread(pic, 256);
	else if (threads_num == 0)
		max_levels = calc_max_levels_default(256, pic, 256);
	else
		max_levels = calc_max_levels_multithread(threads_num, pic,
256);
	apply_levels(pic, max_levels, { 0, 84, 170, 255 });
	double finish = omp_get_wtime();

	unwrap_or_print_error(write_pgm(writer, pic));
```

```
	cout << "Time (" << threads_num << " thread(s)): " << (finish - start) *
1000 << " ms\n";

	for (int i = 0; i < max_levels.size(); i++) {
		if (i > 0)
			cout << ' ';
		cout << max_levels[i];
	}
	cout << '\n';
}

int main(int argc, char* argv[])
{
	if (argc != 4) {
#ifdef _DEBUG
		char* test[] = {
			"",
			"0",

"C:/Users/JazzM/source/repos/LabOpenMp/LabOpenMp/test_data/in.pgm",

"C:/Users/JazzM/source/repos/LabOpenMp/LabOpenMp/test_data/out.pgm"
		};
		argc = 4;
		argv = test;
#else
		cerr << "Wrong number of arguments.";
		return 0;
#endif
	}

	noexcept_reader reader;
	unwrap_or_print_error(reader.open(argv[2]));
	noexcept_writer writer;
	unwrap_or_print_error(writer.open(argv[3]));

	vector<vector<pixel>> pic =
unwrap_or_print_error(parse_pgm(reader));

	normal_start(pic, std::stoi(argv[1]), writer);
	//measure_threads_num(pic);
	//measure_dynamic_blocks(pic);
```

```
        //measure_one_thread(pic);
}
```

measurements.hpp:

```
#pragma once

#include <vector>

#include "utility.hpp"

using std::vector;

void measure_threads_num(vector<vector<pixel>>& pic);
void measure_dynamic_blocks(vector<vector<pixel>>& pic);
void measure_one_thread(vector<vector<pixel>>& pic);
```

noexcept_io.hpp:

```
#pragma once

#include <string>
#include <vector>
#include <fstream>
#include <stdint.h>
#include <iostream>

#include "utility.hpp"

using std::string;

namespace noexcept_io {
    const int BYTE_SIZE = 8;

    class noexcept_stream {
    protected:
    public:
        std::fstream stream;
        virtual ~noexcept_stream();
        virtual maybe_error open(const string& filename) noexcept = 0;
```

```
    };

    class noexcept_reader : noexcept_stream {
        std::ifstream stream;
        std::vector<char> raw;
        int cur_ind = 0;
    public:
        maybe_error open(const string& filename) noexcept override;

        or_error<char> read_char() noexcept;
        or_error<int> read_int() noexcept;
        void skip_spaces() noexcept;
    };

    class noexcept_writer : noexcept_stream {
        std::ofstream stream;
    public:
        maybe_error open(const string& filename) noexcept override;

        template<typename T>
        maybe_error write(T val) noexcept {
            if (!(stream << val).good())
                return error("Couldn't write requeted type.");
            return ok();
        }

        template<typename T>
        maybe_error writeln(T val) noexcept {
            if (!(stream << val).good())
                return error("Couldn't write requeted type.");
            if (!stream.put(char(10)).good())
                return error("Couldn't write next line.");
            return ok();
        }
    };
}
```

otsu.hpp:

```
#pragma once
```

```
#include <vector>

#include "utility.hpp"

using std::vector;

vector<int> calc_max_levels_one_thread(const vector<vector<pixel>>& pic,
int pic_dim);
vector<int> calc_max_levels_multithread(int threads, const
vector<vector<pixel>>& pic, int pic_dim);
vector<int> calc_max_levels_default(int dynamic_blocks, const
vector<vector<pixel>>& pic, int pic_dim);
void apply_levels(vector<vector<pixel>>& pic, const vector<int>& levels,
const vector<pixel>& maps_to);
```

pgm_parsing.hpp:

```
#include "utility.hpp"
#include "noexcept_io.hpp"

using namespace noexcept_io;

or_error<vector<vector<pixel>>> parse_pgm(noexcept_reader& reader);
maybe_error write_pgm(noexcept_writer& writer, vector<vector<pixel>>&
pic);
```

utility.hpp:

```
#pragma once

#include <variant>
#include <string>

using std::string;
using pixel = uint8_t;

struct error {
        string text;
        explicit error (string s) {
                text = s;
```

```
	}
};

template<typename T>
using or_error = std::variant<T, error>;

using maybe_error = or_error<std::monostate>;
using ok = std::monostate;

template<typename T>
bool is_error(const or_error<T>& value) noexcept {
	return std::holds_alternative<error>(value);
}

template<typename T>
error get_error(const or_error<T>& value) noexcept {
	return std::get<error>(value);
}

template<typename T>
T get_value(const or_error<T>& value) noexcept {
	return std::get<T>(value);
}

template<typename T>
bool unwrap_to_or_error(const or_error<T>& value, T& dest) noexcept {
	if (is_error(value))
		return true;
	dest = get_value(value);
	return false;
}

template<typename T>
T unwrap_or_print_error(const or_error<T>& value) noexcept {
	if (is_error(value)) {
		std::cerr << get_error(value).text;
		exit(0);
	}
	return get_value(value);
}
```

measurements.cpp:

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

#include "measurements.hpp"
#include "otsu.hpp"

using std::cout;
using std::pair;

void measure_threads_num(vector<vector<pixel>>& pic) {
	vector<pair<int, double>> res;
	for (int thread_num = 1; thread_num <= 32; thread_num <<= 1) {
		volatile double total_time = 0;

		for (int i = 0; i < 30; i++) {
			double start = omp_get_wtime();
			vector<int> max_levels =
calc_max_levels_multithread(thread_num, pic, 256);
			apply_levels(pic, max_levels, { 0, 84, 170, 255 });
			double finish = omp_get_wtime();
			total_time += finish - start;
		}
		res.emplace_back(thread_num, total_time / 30 * 1000);
	}
	for (auto e : res)
		cout << e.first << ", ";
	cout << '\n';
	for (auto e : res)
		cout << e.second << ", ";
}

void measure_dynamic_blocks(vector<vector<pixel>>& pic) {
	vector<pair<int, double>> res;
	for (int dynamic_blocks = 1; dynamic_blocks <= 512; dynamic_blocks
<<= 1) {
		volatile double total_time = 0;

		for (int i = 0; i < 30; i++) {
			double start = omp_get_wtime();
			vector<int> max_levels =
calc_max_levels_default(dynamic_blocks, pic, 256);
```

```
                        apply_levels(pic, max_levels, { 0, 84, 170, 255 });
                        double finish = omp_get_wtime();
                        total_time += finish - start;
                }
                res.emplace_back(dynamic_blocks, total_time / 30 * 1000);
        }
        for (auto e : res)
                cout << e.first << ", ";
        cout << '\n';
        for (auto e : res)
                cout << e.second << ", ";
}

void measure_one_thread(vector<vector<pixel>>& pic) {
        volatile double total_time = 0;

        for (int i = 0; i < 30; i++) {
                double start = omp_get_wtime();
                vector<int> max_levels = calc_max_levels_one_thread(pic,
256);
                apply_levels(pic, max_levels, { 0, 84, 170, 255 });
                double finish = omp_get_wtime();
                total_time += finish - start;
        }
        cout << total_time / 30 * 1000;
}
```

no_except.cpp:

```
#include "noexcept_io.hpp"

using std::monostate;

namespace noexcept_io {
   noexcept_stream::~noexcept_stream() {
      stream.close();
   }

   maybe_error noexcept_reader::open(const string& filename) noexcept {
      stream.open(filename, std::ifstream::in | std::ifstream::binary);
      if (!stream.is_open())
```

```
            return error("Couldn't open the input file.");
        raw = std::vector<char>(std::istreambuf_iterator<char>(stream),
std::istreambuf_iterator<char>());
        return ok();
    }

    or_error<char> noexcept_reader::read_char() noexcept {
        if (cur_ind >= raw.size())
            return error("Couldn't read byte: EOF.");
        return raw[cur_ind++];
    }

    or_error<int> noexcept_reader::read_int() noexcept {
        while (cur_ind < raw.size() && (raw[cur_ind] < '0' || raw[cur_ind] > '9'))
            cur_ind++;
        if (cur_ind == raw.size())
            return error("Couldn't read int: EOF.");
        int res = 0;
        while (cur_ind < raw.size() && raw[cur_ind] >= '0' && raw[cur_ind] <=
'9')
            res = res * 10 + raw[cur_ind++] - '0';
        return res;
    }

    void noexcept_reader::skip_spaces() noexcept {
        while (cur_ind < raw.size() && std::isspace(static_cast<unsigned
char>(raw[cur_ind])))
            cur_ind++;
    }

    maybe_error noexcept_writer::open(const string& filename) noexcept {
        stream.open(filename, std::ofstream::out | std::ifstream::binary);
        if (!stream.is_open())
            return error("Couldn't open the output file.");
        return ok();
    }
}
```

otsu.cpp:

```
#include <iostream>
```

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

#include "otsu.hpp"

long long safe(long long num) {
	if (num == 0)
		return 1;
	return num;
}

double get_disp_sum(
	const vector<long long>& pref_sum_in_pic,
	const vector<long long>& freq_pref_sum,
	int pic_dim,
	int lev0,
	int lev1,
	int lev2
) {
	return (double)pref_sum_in_pic[lev0] * pref_sum_in_pic[lev0] /
		safe(freq_pref_sum[lev0]) +
		(double)(pref_sum_in_pic[lev1] - pref_sum_in_pic[lev0]) *
(pref_sum_in_pic[lev1] - pref_sum_in_pic[lev0]) /
		safe(freq_pref_sum[lev1] - freq_pref_sum[lev0]) +
		(double)(pref_sum_in_pic[lev2] - pref_sum_in_pic[lev1]) *
(pref_sum_in_pic[lev2] - pref_sum_in_pic[lev1]) /
		safe(freq_pref_sum[lev2] - freq_pref_sum[lev1]) +
		(double)(pref_sum_in_pic[pic_dim - 1] - pref_sum_in_pic[lev2])
* (pref_sum_in_pic[pic_dim - 1] - pref_sum_in_pic[lev2]) /
		safe(freq_pref_sum[pic_dim - 1] - freq_pref_sum[lev2]);
}

void calc_pref_sums(
	vector<long long>& freq,
	vector<long long>& freq_pref_sum,
	vector<long long>& pref_sum_in_pic
) {
	freq_pref_sum = freq;
	freq_pref_sum[0] = freq[0];
	for (int pix = 1; pix < freq.size(); pix++)
		freq_pref_sum[pix] += freq_pref_sum[pix - 1];
```

```
	pref_sum_in_pic.resize(freq.size(), 0);
	pref_sum_in_pic[0] = freq[0];
	for (int pix = 1; pix < freq.size(); pix++)
		pref_sum_in_pic[pix] = pref_sum_in_pic[pix - 1] + freq[pix] *
(pix + 1);
}

vector<int> calc_max_levels_one_thread(const vector<vector<pixel>>& pic,
int pic_dim) {
	vector<long long> freq(pic_dim, 0);
	for (int row = 0; row < pic.size(); row++)
		for (int col = 0; col < pic[0].size(); col++)
			freq[pic[row][col]]++;

	vector<long long> freq_pref_sum, pref_sum_in_pic;
	calc_pref_sums(freq, freq_pref_sum, pref_sum_in_pic);

	vector<int> max_levels = { 0, 1, 2 };
	double max_disp = 0;
	for (int lev0 = 0; lev0 < pic_dim - 2; lev0++) {
		for (int lev1 = lev0 + 1; lev1 < pic_dim - 1; lev1++) {
			for (int lev2 = lev1 + 1; lev2 < pic_dim; lev2++) {
				double disp = get_disp_sum(pref_sum_in_pic,
freq_pref_sum, pic_dim, lev0, lev1, lev2);
				if (disp > max_disp) {
					max_disp = disp;
					max_levels = { lev0, lev1, lev2 };
				}
			}
		}
	}
	return max_levels;
}

vector<int> calc_max_levels_multithread(int threads, const
vector<vector<pixel>>& pic, int pic_dim) {
	vector<long long> freq(pic_dim, 0);
#pragma omp parallel for num_threads(threads)
	for (int cell = 0; cell < pic.size() * pic[0].size(); cell++) {
		int row = cell / pic[0].size();
		int col = cell % pic[0].size();
#pragma omp atomic
```

```
                freq[pic[row][col]]++;
        }

        vector<long long> freq_pref_sum, pref_sum_in_pic;
        calc_pref_sums(freq, freq_pref_sum, pref_sum_in_pic);

        vector<int> max_levels = { 0, 1, 2 };
        double max_disp = 0;
        int mx_sz = pic_dim * pic_dim * pic_dim;

#pragma omp parallel num_threads(threads)
        {
                vector<int> max_levels_local = { 0, 1, 2 };
                double max_disp_local = 0;
#pragma omp for
                for (int x = 0; x < mx_sz; x++) {
                        int lev0 = x % pic_dim;
                        int lev1 = (x / pic_dim) % pic_dim;
                        int lev2 = x / pic_dim / pic_dim;
                        if (lev1 <= lev0 || lev2 <= lev1)
                                continue;
                        double disp = get_disp_sum(pref_sum_in_pic,
freq_pref_sum, pic_dim, lev0, lev1, lev2);
                        if (disp > max_disp_local) {
                                max_disp_local = disp;
                                max_levels_local = { lev0, lev1, lev2 };
                        }
                }
#pragma omp critical
                {
                        if (max_disp_local > max_disp) {
                                max_disp = max_disp_local;
                                max_levels = max_levels_local;
                        }
                }
        }
        return max_levels;
}

vector<int> calc_max_levels_default(int dynamic_blocks, const
vector<vector<pixel>>& pic, int pic_dim) {
        vector<long long> freq(pic_dim, 0);
```

```
#pragma omp parallel for
	for (int cell = 0; cell < pic.size() * pic[0].size(); cell++) {
		int row = cell / pic[0].size();
		int col = cell % pic[0].size();
#pragma omp atomic
		freq[pic[row][col]]++;
	}

	vector<long long> freq_pref_sum, pref_sum_in_pic;
	calc_pref_sums(freq, freq_pref_sum, pref_sum_in_pic);

	vector<int> max_levels = { 0, 1, 2 };
	double max_disp = 0;
	int mx_sz = pic_dim * pic_dim * pic_dim;

#pragma omp parallel
	{
		vector<int> max_levels_local = { 0, 1, 2 };
		double max_disp_local = 0;
#pragma omp for schedule(dynamic, dynamic_blocks)
		for (int x = 0; x < mx_sz; x++) {
			int lev0 = x % pic_dim;
			int lev1 = (x / pic_dim) % pic_dim;
			int lev2 = x / pic_dim / pic_dim;
			if (lev1 <= lev0 || lev2 <= lev1)
				continue;
			double disp = get_disp_sum(pref_sum_in_pic,
freq_pref_sum, pic_dim, lev0, lev1, lev2);
			if (disp > max_disp_local) {
				max_disp_local = disp;
				max_levels_local = { lev0, lev1, lev2 };
			}
		}
#pragma omp critical
		{
			if (max_disp_local > max_disp) {
				max_disp = max_disp_local;
				max_levels = max_levels_local;
			}
		}
	}
	return max_levels;
```

```
}

void apply_levels(vector<vector<pixel>>& pic, const vector<int>& levels,
const vector<pixel>& maps_to) {
	for (auto& row : pic)
		for (auto& pix : row)
			pix = maps_to[lower_bound(levels.begin(), levels.end(),
pix) - levels.begin()];
}
```

pgm_parsing.cpp:

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

#include "otsu.hpp"
#include "pgm_parsing.hpp"

using std::to_string;
using namespace noexcept_io;

or_error<vector<vector<pixel>>> parse_pgm(noexcept_reader& reader) {
	auto tmp = reader.read_char();
	char c1;
	if (unwrap_to_or_error(tmp, c1))
		return get_error(tmp);

	tmp = reader.read_char();
	char c2;
	if (unwrap_to_or_error(tmp, c2))
		return get_error(tmp);

	if (c1 != 'P' || c2 != '5')
		return error("Wrong file format: there is no P5 in the
beginning.");

	int width;
	auto tmp1 = reader.read_int();
	if (unwrap_to_or_error(tmp1, width))
```

```
		return get_error(tmp1);

	int height;
	tmp1 = reader.read_int();
	if (unwrap_to_or_error(tmp1, height))
		return get_error(tmp1);

	int dim;
	tmp1 = reader.read_int();
	if (unwrap_to_or_error(tmp1, dim))
		return get_error(tmp1);
	if (dim != 255)
		return error("Wrong file format: given pixel dimention is not
supported.");

	reader.skip_spaces();

	vector<vector<pixel>> pic(height, vector<pixel>(width, 0));
	for (int r = 0; r < height; r++) {
		for (int c = 0; c < width; c++) {
			auto pix = reader.read_char();
			char ch;
			if (unwrap_to_or_error(pix, ch))
				return get_error(pix);
			if (ch < 0)
				pic[r][c] = (int)ch + 256;
			else
				pic[r][c] = ch;
		}
	}
	return pic;
}

maybe_error write_pgm(noexcept_writer& writer, vector<vector<pixel>>&
pic) {
	auto tmp = writer.writeln("P5");
	if (is_error(tmp))
		return get_error(tmp);
	tmp = writer.writeln(to_string(pic.size()) + " " +
to_string(pic[0].size()));
	if (is_error(tmp))
		return get_error(tmp);
```

```
	tmp = writer.writeln("255");
	if (is_error(tmp))
		return get_error(tmp);
	for (auto& row : pic) {
		for (auto& pix : row) {
			auto tmp = writer.write(pix);
			if (is_error(tmp))
				return get_error(tmp);
		}
	}
	return ok();
}
```