Лабораторная работа №4	M3137	2023
OpenMP	Нартов Дмитрий Николаевич	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: C++ 20, clang 14.0.5, Cmake, OpenMP 2.0.

Описание

Реализовать программу выполняющую пороговую фильтрации изображения методом Оцу для 3 порогов. В процессе реализации использовать конструкции OpenMP для многопоточного программирования.

Вариант

hard

Конструкции OpenMP.

OpenMp - это API для распараллеливания программ, которая руководствуется принципом fork-join, то есть изначально у нас есть один главный поток, который исполняет программу, а при попадании в блок, который должен исполняться параллельно, он создает новые потоки, которые по окончанию параллельного блока уничтожаются, и продолжается выполнение главным потоком.

Почти все конструкции OpenMP начинаются с директивы #pragma, которая представляет собой указание препроцессору. #pragma omp parallel создает блок, который будет распараллелен, также можно дописать for и тогда это сократит запись и нам не придется писать в блоке parallel дополнительно #pragma omp for. Немаловажным элементом в записи parallel является if с условием, при выполнении которого директива будет исполнена, в противном случае нет. Директивы с for поддерживают команду schedule, с помощью которой настраивается разделение итераций между потоками. #pragma critical отвечает за блок, который будет единовременно исполняться, только одним потоком. Это нужно, чтобы избежать состояния гонки. Стоит упомянуть и о ключевом слове nowait, которое может использоваться в блоке parallel. Это слово означает, что мы не ждем пока закончат работу другие потоки, а идем исполнять следующие команды.

Кроме директив существуют и методы, которые выполняют не менее важную работу. omp_set_dynamic(int) включает динамическое выделение потоков, если задано ненулевое число, выключает иначе. Динамическое выделение потоков работает следующим образом: у программы есть доступное число потоков, но для каждого блока parallel программно решается отдельно какое именно количество потоков использовать.

omp_set_num_threads(int) позволяет поставить максимально допустимое количество потоков. omp_get_wtime() выдает время в секундах начиная с момента запуска программы.

Рассмотрим детальнее конструкции, использующиеся в моей программе:

```
#pragma omp parallel if (openmp_switch)
```

Эта конструкция создает блок parallel, только если значение переменной openmp_switch истинно.

```
#pragma omp for schedule(static)
```

Данная директива, позволяет распараллелить цикл for, и ставит с помощью schedule распределение (static, dynamic и т.д., а также chunk_size). Static равномерно распределяет итерации между потоками, то есть делит общее количество итераций на количество потоков и отдает первому потоку блок итераций размером с полученное значение, начиная с 1 итерации, второму - блок такой же длины, начиная с конца первого и т. д. dynamic - тоже раздает итерации последовательными блоками, только распределение идет динамически, то есть блок, который сейчас находится в очереди отдается первому свободному потоку. chunk_size отвечает за размер блока. Тут распределение уже интереснее. Со static, каждому потоку дадут блок размером chunk_size и когда все потоки закончат выполнение, только тогда начнется новое распределение блоков. dynamic будет отдавать новые блоки освободившимся потокам, то есть не будет ждать остальных для распределения.

```
#pragma omp critical
```

Это директива говорит, что эта часть кода выполняется одновременно только одним потоком, а остальные ждут своей очереди для выполнения этого блока.

```
#pragma omp for nowait schedule(static)
```

Эта директива похожа на то, что было описано ранее, только здесь есть еще ключевое слово nowait, которое, позволяет свободным потокам (которые вообще не участвуют в выполнении цикла) пропустить этот цикл и исполнять следующие команды.

Описание программы.

Программа очень проста, она состоит из ввода исходной картинки, применения к ней алгоритма фильтрации и вывода.

Функция input производит открытие входного файла и ввод его в программу. Для этого создается ifstream и с помощью него происходит чтение. Также проводится несколько проверок: если возникают какие проблемы с открытием и чтением файла, то методы ifstream выбросят исключение, которое мы поймаем в main, так как вызов этой функции облачен в блок try-catch, если же нам подадут файл, который не является pgm картинкой, то мы выведем сообщение об ошибки и закончим выполнение программы.

Функция output выводит картинку в выходной файл. Здесь таким же образом обрабатываются ошибки открытия, создания и записи, как и в функции чтения. Для выполнения этой функции используется ofstream.

process_image - это функция, отвечающая за обработку изображения. Здесь и написан весь алгоритм пороговой фильтрации методом Оцу. Сначала мы подсчитываем гистограмму цветов. Этот подсчет распараллелен и логика здесь следующая: для каждого потока мы создаем свою гистограмму, которую подсчитываем, а потом в блоке critical объединяем все эти гистограммы в одну, которая уже содержит информацию про всю картинку. Далее мы подсчитываем префиксные суммы, которые позволят нам за O(1) получать сумму на отрезке. Наконец три вложенных цикла, которые производят поиск наилучших порогов. Внешний цикл мы распараллеливаем, используя #pragma omp for, в самих внутренностях цикла производится расчет лучших порогов для каждого треда, а потом эти результаты сравниваются в блоке critical и выбираются

лучшие значения порогов. В конце концов, мы производим пересчет картинки заменяя цвета на новые в соответствии со значениями порогов. Конечно же, этот цикл тоже стоит распараллелить, ведь здесь мы каждый пиксель картинки пересчитываем отдельно, что не создает никаких проблем для реализации многопоточности.

Безусловно, нужно реализовать некоторую обработку ошибок пользователя, поэтому нужно проверить, что нам ввели 3 аргумента через командную строку, что количество тредов - это число удовлетворяющее критериям описанным в тз, ну и облачить запуски функций input и output в блок try-catch.

Результат работы.

Процессор: AMD Ryzen 3 3200U. 2 ядра, 4 потока.

Входная картинка:

Выходная картинка:

Лог программы:

```
[nrtv@tomasshelbylab test_lab_4]$ ./hard 4 in.pgm out.pgm
77 130 187
Time (4 thread(s)): 17.7202 ms
```

Команда компиляции:

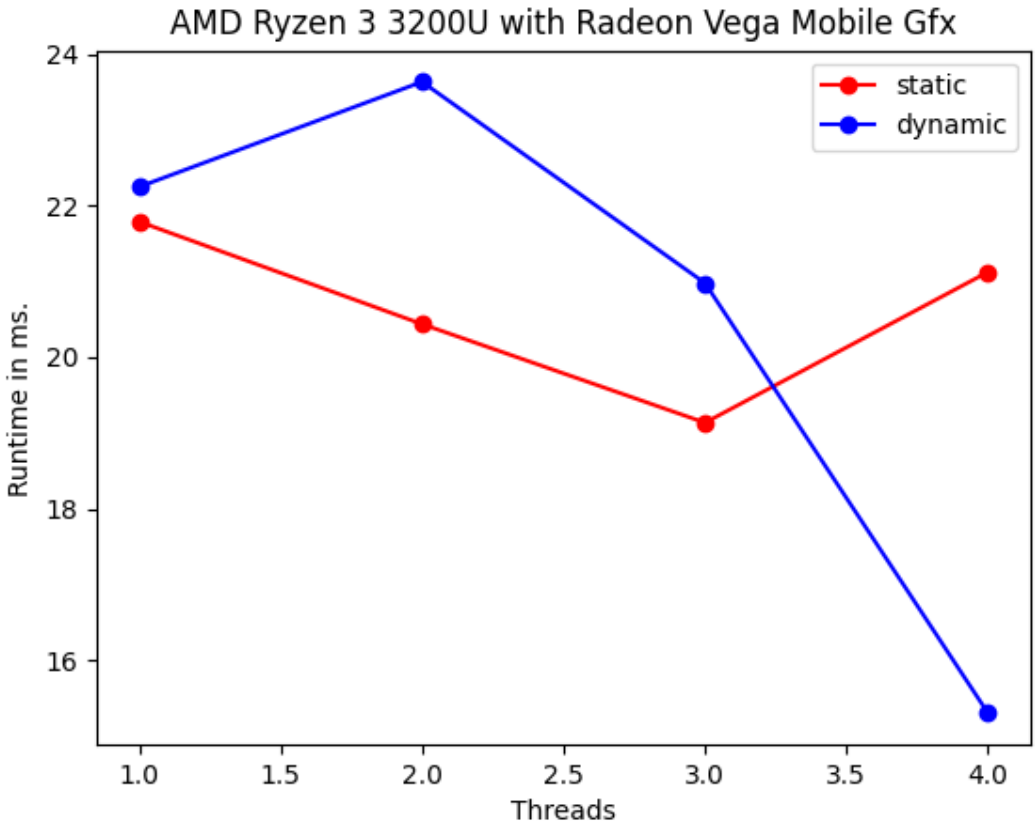
```
[nrtv@tomasshelbylab test_lab_4]$ clang++ -std=c++20 -Ofast -o hard -fopenmp
hard.cpp
```

Анализ и сравнение результатов.

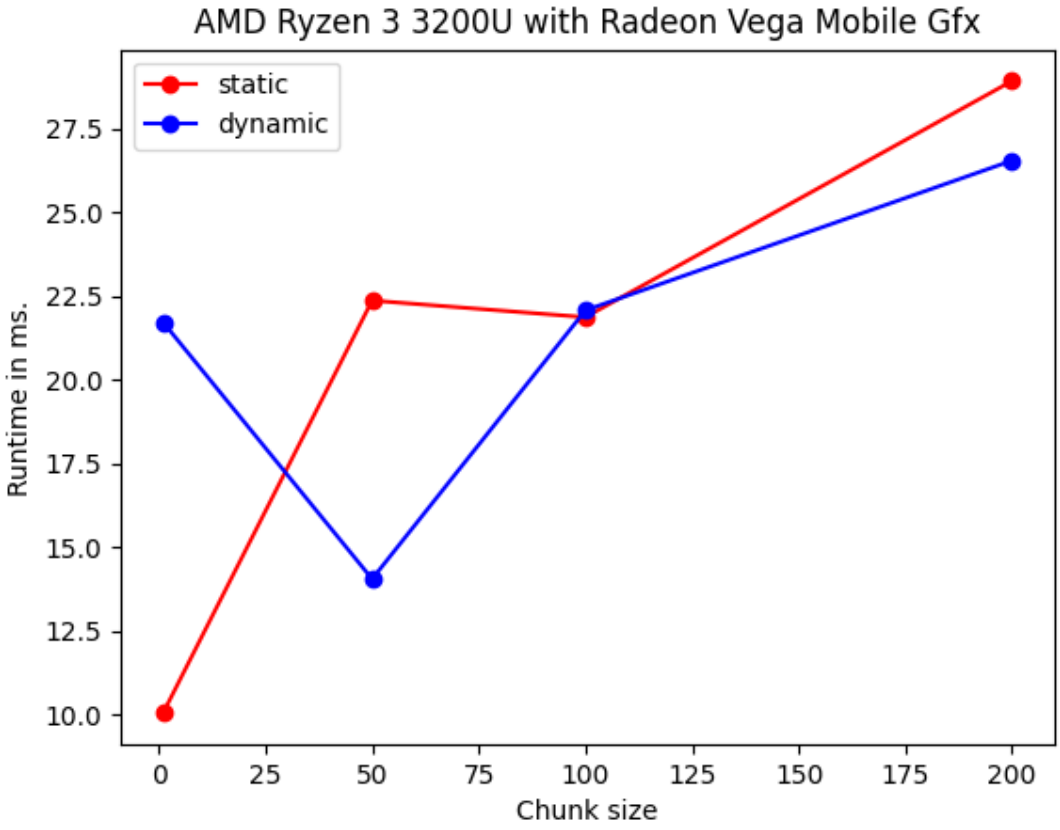
Мне стало интересно и я решил протестировать программу на разных процессорах, а затем сравнить результаты. Мне удалось воспользоваться следующими процессорами: AMD Ryzen 3 3200U, AMD Ryzen 5 4600H, AMD Ryzen 7 5800H, Intel core i7-1165G7. При тестировании использовался скрипт на python. Для построения графиков использовалась библиотека matplotlib. В первом графике, для каждого числа потоков программа запускалась 50 раз и бралось среднее арифметическое времен. Во втором графике для каждого набора static/dynamic и chunk_size (значения которых всегда были 1, 50, 100, 200) программа запускалась 50 раз и бралось среднее арифметическое времен. Для третьего графика производилась 1000 запусков для выключенного OpenMP и нет и бралось среднее арифметическое времен.

AMD Ryzen 3 3200U.

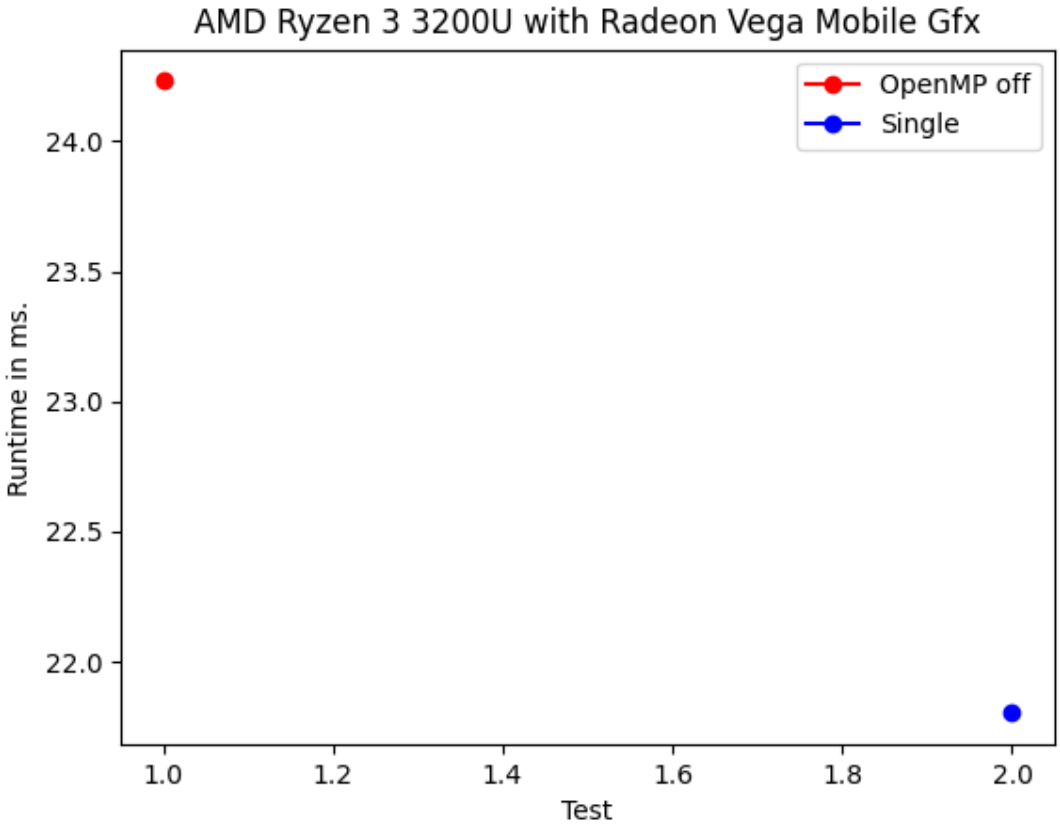
Данный процессор не особо мощный. Он имеет всего 2 ядра и 4 потока.



На данном процессоре на 1-3 тредах задача быстрее выполняется со static. Скорее всего dynamic, здесь медленнее, так как мы тратим больше времени на распределение задач и синхронизацию потоков. Но на 4 тредах dynamic довольно сильно выигрывает static. Тут уже параллельное исполнение кода дает выигрыш, хотя мы все еще тратим время на распределение и синхронизацию.



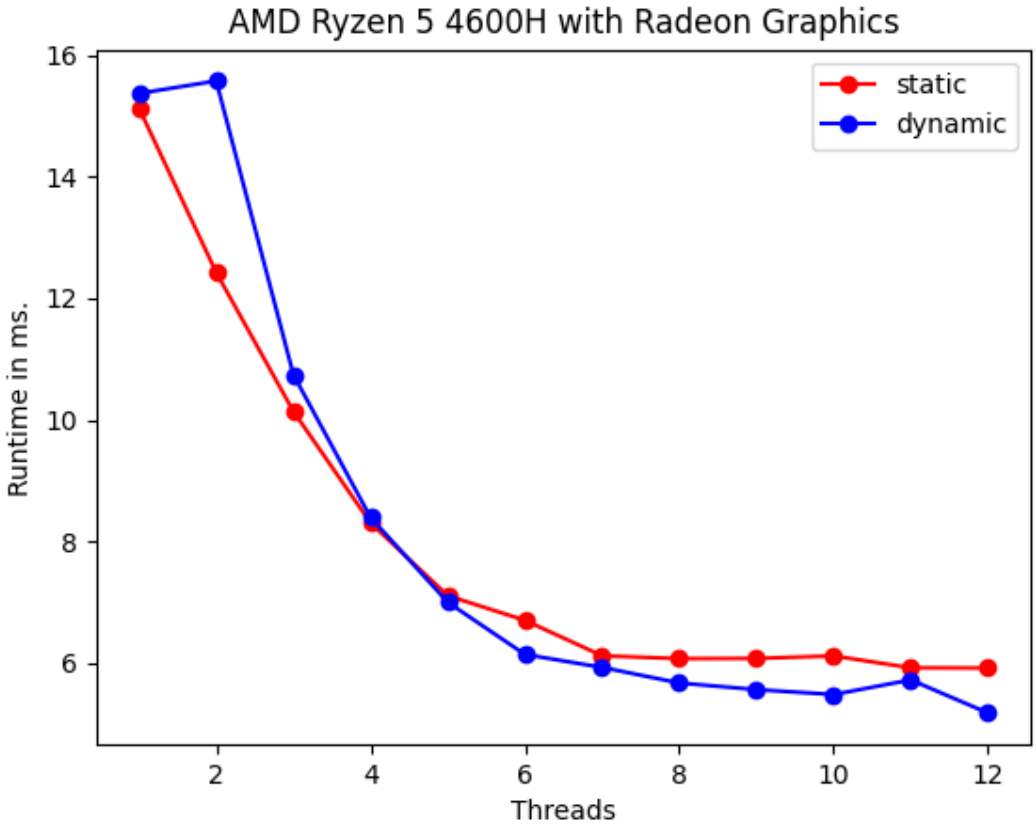
Тут мы имеем интересный результат. На chunk_size равном 1, static быстрее потому, что ожидание тредов почти не занимает времени, а вот dynamic, возможно тратит время на выбор потока и в этом момент другие потоки образуют очередь, из-за чего программа выполняется медленнее. На chunk_size равном 50, выигрывает dynamic, так как тут динамическое разделение задач между потоками выполняется эффективнее. На следующих значения уже chunk_size становится менее оптимальным, и нам приходиться тратить время на ожидание других потоков(это про dynamic).



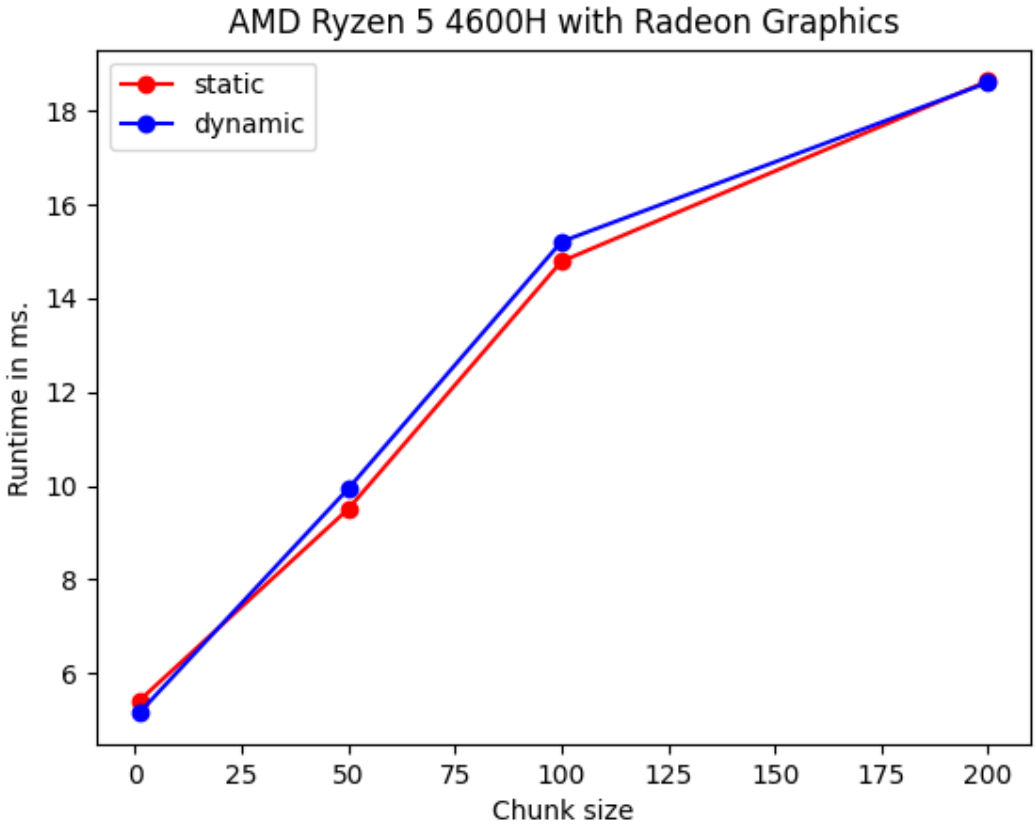
Тут тоже интересный результат, так как программа без OpenMP была медленнее хотя и всего на 2 мс. Скорее всего это связано, с фоновыми процессами на компьютере, так как разница небольшая.

AMD Ryzen 5 4600H.

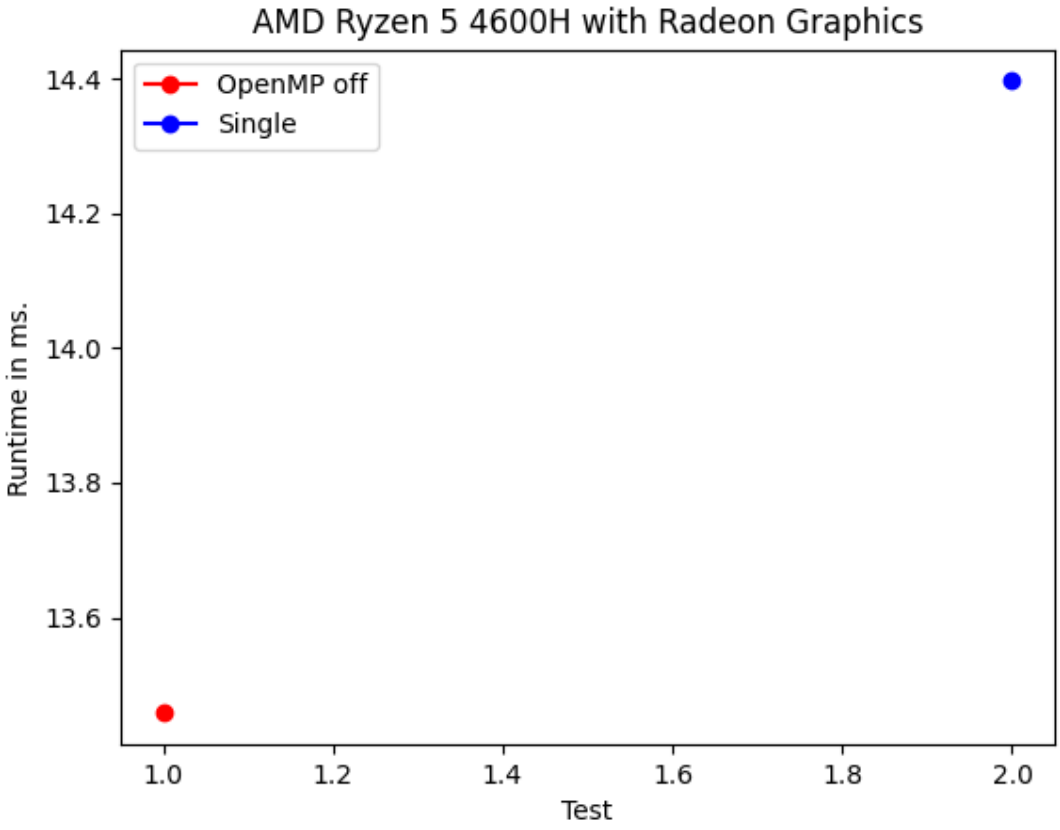
Этот процессор уже лучше. Здесь мы имеем 6 ядер и 12 потоков.



Здесь нужно обратить внимание на несколько деталей. Так же как и на первом процессоре у нас dynamic сильно проигрывает static на 2 ядрах, причины этого я описал выше. dynamic начинает выигрывать начиная с 6 потоков. Тут, наверное, лучше всего начинает работать динамическое распределение.



Второй график не сильно похож на соответствующих график для первого процессора. Тут на chunck_size 1 быстрее dynamic и, скорее всего, это связано с фоновыми процессами на компьютере. Далее все почти так же, но dynamic немного быстрее, потому что процессор имеет больше потоков, для распределения задач между ними.



Тут картина схожая с первым процессором. Разница небольшая, но выигрывает программа с отключенным OpenMP, что в целом логично, так не производятся никакие манипуляции OpenMP.

AMD Ryzen 7 5800H.

Процессор хоть и мобильный, но достаточно мощный. 8 ядер, 16 потоков.

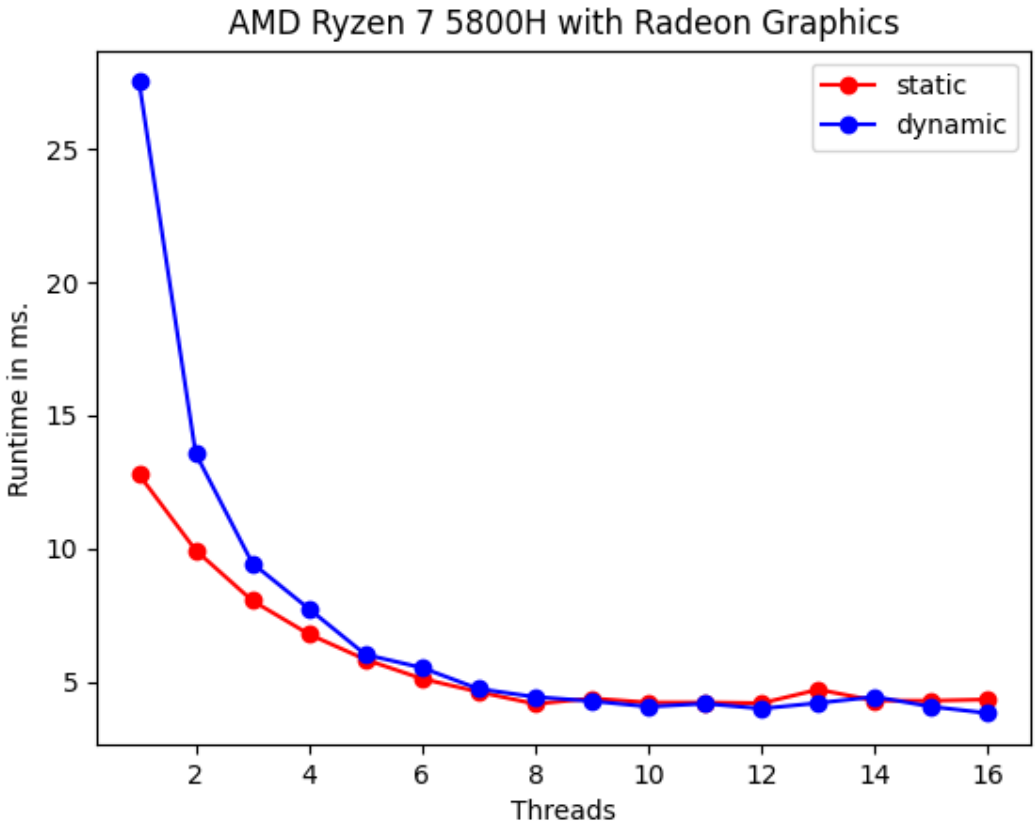
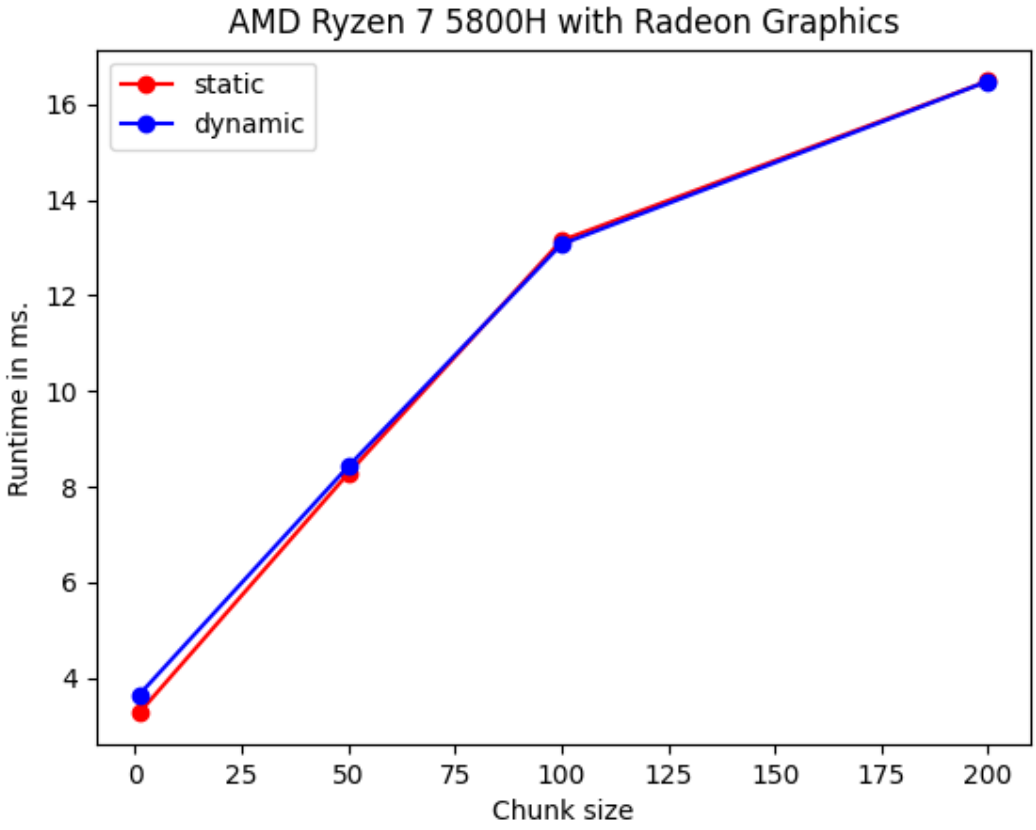
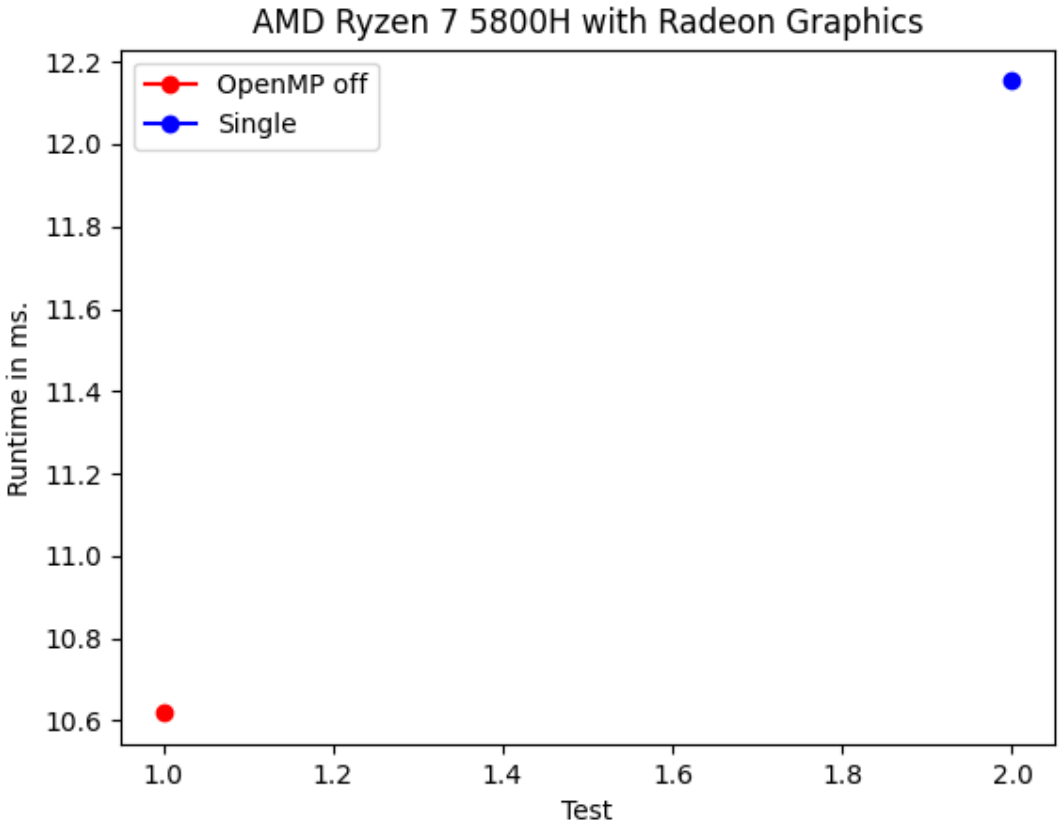


График похож на соответствующий для 2 процессора, только dynamic начинает выигрывать только на 9 потоках. Выигрыш static на первых порах, по той же причине, что и раньше. Выигрыш static на 14 потоках связан, наверно, с малой выборкой.



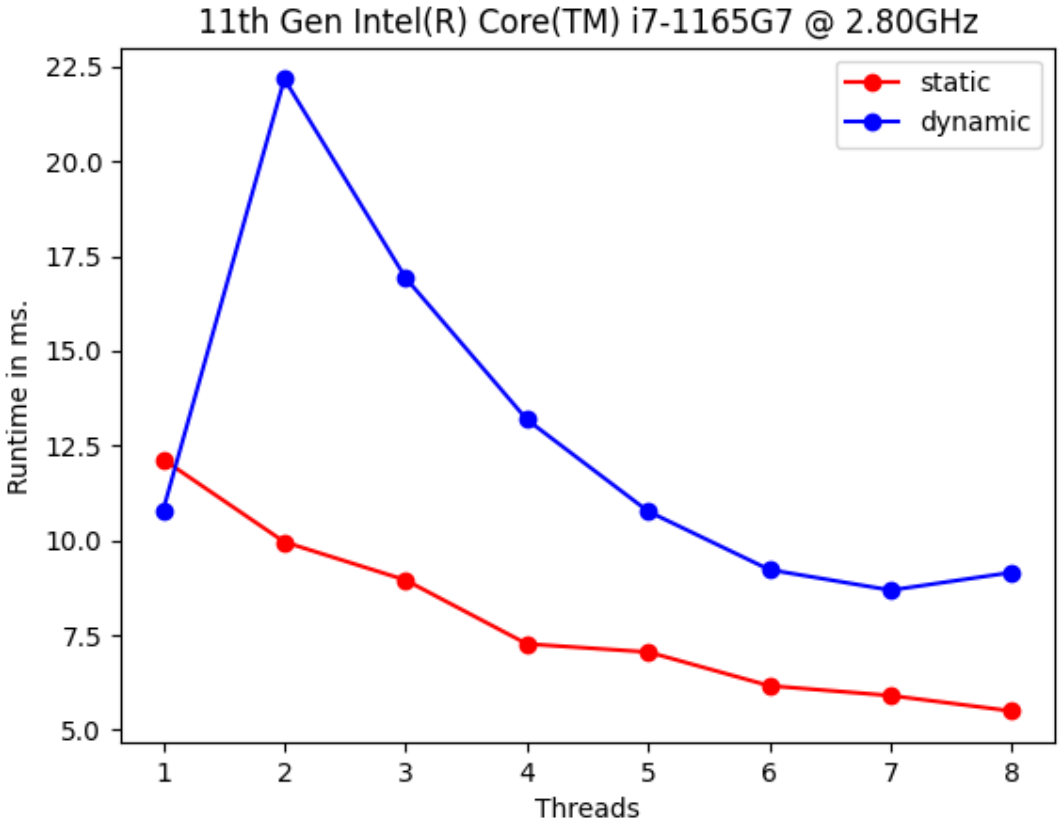
Результат, точно такой же как и на 2 процессоре, так что и описывать нечего.



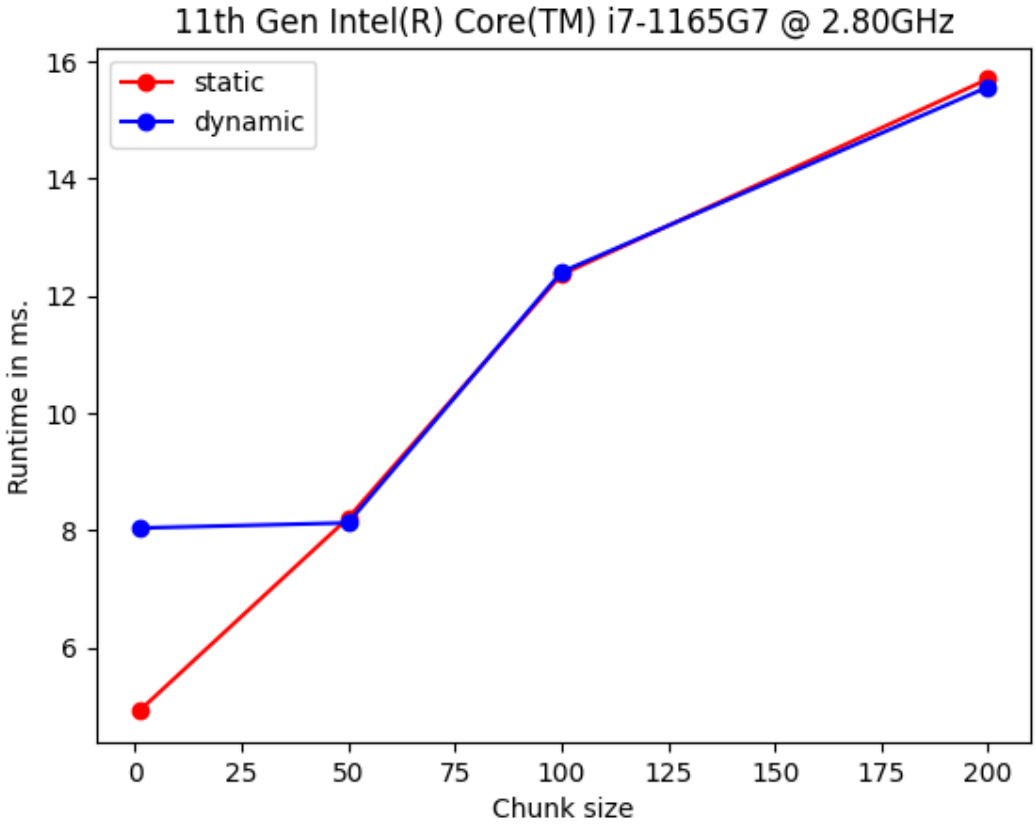
Результат на лицо, и сказать нечего. Результат, такой же был на предыдущем процессоре.

Intel core i7-1165G7.

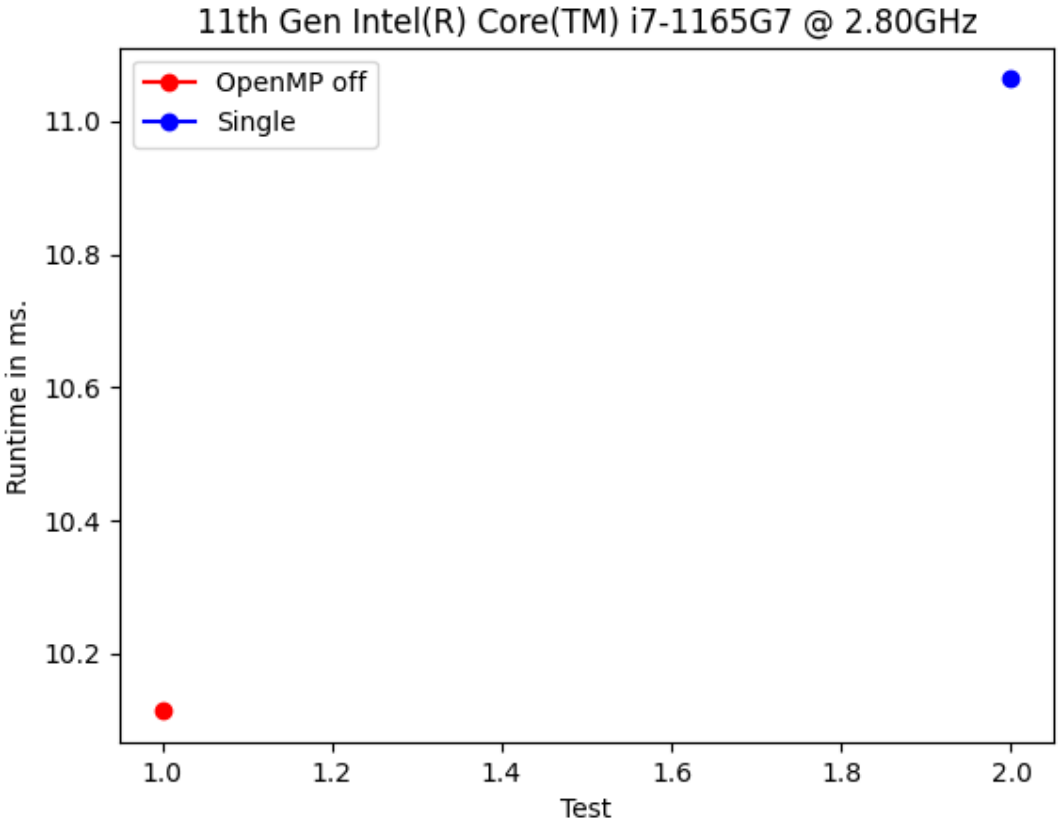
Наконец-то процессор не AMD. Это мобильный процессор 11 поколения. 4 ядра, 8 потоков.



Тут имеем совсем другой результат. Видимо, из-за частоты, либо модели исполнения процессора мы получаем на static существенный выигрыш. Резкий прыжок на 2 потоках, скорее всего, также зависит от неэффективности распределения и синхронизации.



Тут уже что-то более похожее на предыдущие варианты. static выигрывает на chunk_size 1, все по той же причине описанной на предыдущем графике. Но dynamic, всё же, самую малость быстрее на максимальном chunck_size, видимо, здесь начинает работать динамическое распределение.



Тут уже знакомая картина. Быстрее отработало с выключенным OpenMP, что закономерно всё по тем же причинам.

Выводы.

Некоторые результаты получились довольно неожиданными. Но можно, все-таки, выделить общие черты. Почти у всех процессоров, dynamic работает лучше на большом количестве тредов. Конечно же, на скорость выполнения влияет много факторов, но в большинстве случаев dynamic оказывается быстрее на максимальном количестве потоков. Почти всегда, при увеличении chunk_size программа замедлялась. Ну и в большинстве случаев программы быстрее выполнялась на одном потоке быстрее с выключенным OpenMP.

Список источников.

1. https://www.openmp.org//wp-content/uploads/cspec20.pdf
2. https://drive.google.com/file/d/122YfkZEbzzHFtoSx-LwwUHpg7rmqiNVw/view
3. https://drive.google.com/file/d/15osD8y6aRGiaAccpddzhTzcvfe6zrSC3/view

Листинг.

hard.cpp

```
#include <omp.h>

#include <array>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

unsigned int width, height, depth;
std::vector<unsigned char> picture;
bool openmp_switch;  // 1 - on 0 - off

void input(const std::filesystem::path &input_file_name) {
    std::ifstream in = std::ifstream(input_file_name, std::ios::in |
std::ios::binary);
    in.exceptions(std::ios::failbit);

    std::string sign;
    in >> sign;
    if (sign != "P5") {
        std::cout << "Unsupported file format" << std::endl;
        exit(-1);
    }
    in >> width >> height >> depth;
    in.ignore(1);
    picture.resize(height * width);
    in.read(reinterpret_cast<char *>(picture.data()), width * height);
}

void output(const std::filesystem::path &output_file_name) {
    std::ofstream out = std::ofstream(output_file_name, std::ios::out |
std::ios::binary);
    out << "P5\n"
        << width << ' ' << height << '\n'
        << depth << '\n';
    for (const auto &i : picture) {
        out << i;
    }
}

inline void process_image() {
    std::array<unsigned int, 256> histogram{};

#pragma omp parallel if (openmp_switch)
    {
        std::array<unsigned int, 256> thread_histogram{};

#pragma omp for schedule(static)
        for (size_t i = 0; i < picture.size(); i++) {
            thread_histogram[picture[i]]++;
        }
```

```
#pragma omp critical
        {
            for (size_t i = 0; i < thread_histogram.size(); i++) {
                histogram[i] += thread_histogram[i];
            }
        }
    }

    std::array<unsigned long long, histogram.size() + 1> sums;
    sums[0] = 0;

    std::array<double, histogram.size() + 1> mul_sums;
    mul_sums[0] = 0;

    for (size_t i = 0; i < histogram.size(); i++) {
        sums[i + 1] = sums[i] + histogram[i];
    }

    for (size_t i = 0; i < histogram.size(); i++) {
        mul_sums[i + 1] = mul_sums[i] + i * histogram[i];
    }

    double best_variance = -1;
    unsigned char best_k1 = 0;
    unsigned char best_k2 = 0;
    unsigned char best_k3 = 0;

#pragma omp parallel if (openmp_switch)
    {
        double best_thread_variance = -1;
        unsigned char best_thread_k1 = 0;
        unsigned char best_thread_k2 = 0;
        unsigned char best_thread_k3 = 0;

#pragma omp for nowait schedule(static)
        for (int k1 = 1; k1 <= 253; k1++) {
            const unsigned long long p1 = sums[k1 + 1] - sums[0];
            if (p1 == 0) {
                continue;
            }
            const double m1 = (mul_sums[k1 + 1] - mul_sums[0]) /
static_cast<double>(p1);

            for (int k2 = k1 + 1; k2 <= 254; k2++) {
                const unsigned long long p2 = sums[k2 + 1] - sums[k1 + 1];
                if (p2 == 0) {
                    continue;
                }
                const double m2 = (mul_sums[k2 + 1] - mul_sums[k1 + 1]) /
static_cast<double>(p2);

                for (int k3 = k2 + 1; k3 <= 255; k3++) {
                    const unsigned long long p3 = sums[k3 + 1] - sums[k2 + 1];
                    if (p3 == 0) {
                        continue;
                    }
                    const double m3 = (mul_sums[k3 + 1] - mul_sums[k2 + 1]) /
static_cast<double>(p3);
```

```
                    const unsigned long long p4 = sums[histogram.size()] -
sums[k3 + 1];
                    if (p4 == 0) {
                        continue;
                    }
                    const double m4 = (mul_sums[histogram.size()] - mul_sums[k3
+ 1]) / static_cast<double>(p4);

                    const double main_m = p1 * m1 + p2 * m2 + p3 * m3 + p4 * m4;
                    const double variance =
                            (p1 * (m1 - main_m) * (m1 - main_m) +
                             p2 * (m2 - main_m) * (m2 - main_m)) +
                            (p3 * (m3 - main_m) * (m3 - main_m) +
                             p4 * (m4 - main_m) * (m4 - main_m));

                    if (variance > best_thread_variance) {
                        best_thread_variance = variance;
                        best_thread_k1 = k1;
                        best_thread_k2 = k2;
                        best_thread_k3 = k3;
                    }
                }
            }
        }

#pragma omp critical
        {
            if (best_variance < best_thread_variance) {
                best_variance = best_thread_variance;
                best_k1 = best_thread_k1;
                best_k2 = best_thread_k2;
                best_k3 = best_thread_k3;
            }
        }
    }

    printf("%u %u %u\n", best_k1, best_k2, best_k3);

#pragma omp parallel for schedule(static) if (openmp_switch)
    for (size_t i = 0; i < picture.size(); i++) {
        if (0 <= picture[i] && picture[i] <= best_k1) {
            picture[i] = 0;
        } else if (best_k1 + 1 <= picture[i] && picture[i] <= best_k2) {
            picture[i] = 84;
        } else if (best_k2 + 1 <= picture[i] && picture[i] <= best_k3) {
            picture[i] = 170;
        } else if (best_k3 + 1 <= picture[i]) {
            picture[i] = 255;
        }
    }
}

int main(int argc, char *argv[]) {
    if (argc != 4) {
        std::cout << "Expected 3 arguments, found " << argc - 1 << '\n';
        return 0;
    }
    const std::filesystem::path input_file_name = argv[2], output_file_name =
```

```
argv[3];

    omp_set_dynamic(0);

    int num_threads;

    try {
        num_threads = std::stoi(argv[1]);
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
        return 0;
    }

    openmp_switch = (num_threads != -1);

    if (num_threads == 0) {
        omp_set_num_threads(omp_get_max_threads());
        num_threads = omp_get_max_threads();
    } else if (num_threads > 0) {
        omp_set_num_threads(num_threads);
    } else if (num_threads == -1){
        num_threads = 1;
    } else {
        std::cout << "Incorrect number of threads" << std::endl;
    }

    try {
        input(input_file_name);
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
        return 0;
    }
    const double start_time = omp_get_wtime();

    process_image();

    const double end_time = omp_get_wtime();

    try {
        output(output_file_name);
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
        return 0;
    }
    printf("Time (%i thread(s)): %g ms\n", num_threads, (end_time - start_time)
* 1000);
    return 0;
}
```