Лабораторная работа №4	38	2023
OpenMP	Ситкина Алена Николаевна	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий: работа выполнена на C++. Компилятор g++ (GCC) 12.2.0. Стандарт OpenMP 2.0.

Описание конструкций OpenMP для распараллеливания команд.

```
#pragma omp parallel
```

Листинг 1 – конструкция parallel

Данная директива (листинг 1) обозначает параллельный регион, то есть область программы, которая может исполняться в несколько потоков параллельно.

```
#pragma omp parallel if (num_of_threads != -1)
```

Листинг 2 – конструкция parallel с условным выражением

Условное выражение (листинг 2) определяет, будет ли параллельный регион исполняться в несколько потоков. Если условие равно false, область кода исполняется без распараллеливания.

```
#pragma omp for nowait
```

Листинг 3 – конструкция for (с nowait)

Данная конструкция (листинг 3) используется внутри параллельного региона перед циклом for и указывает, что итерации это цикла должны быть поделены между существующими тредами. nowait позволяет треду, закончившему свою работу раньше остальных, выполнять код дальше, не дожидаясь, когда другие треды закончат исполнять код цикла. Также в этой конструкции можно использовать parallel (перед for), if (как было описано ранее) и schedule (это отдельный параметр, речь о котором пойдёт далее).

```
#pragma omp for nowait schedule(kind, chunk_size)
```

Листинг 4 – конструкция for schedule

Параметр schedule (листинг 4) позволяет указать, как итерации цикла будут распределены между тредами (корректность программы не должна зависеть от того, какой тред какую итерацию исполняет). Аргумент kind может принимать несколько различных значений, среди которых dynamic и static (по тз нужно было исследовать только эти два значения). Dynamic означает, что итерации делятся на блоки размера chunk_size (по умолчанию chunk_size = 1), один тред выполняет один блок. Когда какой-либо тред

закончит выполнять свой блок итерации, ему будет присвоен новый, пока они не закончатся (последний блок может быть меньше по размеру, чем остальные). Static означает, что блоки итераций размера chunk_size (по умолчанию итерации разбиваются на примерно равные по размеру блоки, по одному блоку на тред) статически назначаются тредам циклическим образом в порядке их нумерации.

```
#pragma omp critical
```

Листинг 5 – конструкция critical

Данная конструкция (листинг 5) указывает, что последующий блок команд должен выполняться одновременно только одним тредом.

```
omp_set_dynamic(num_of_threads)
```

Листинг 6 – функция omp_set_dynamic

Функция из листинга 6 позволяет в run-time скорректировать количество потоков, используемых для выполнения параллельных областей, для наилучшего использования системных ресурсов. При этом для каждого из параллельных регионов во время его выполнения число потоков остается фиксированным (пользователь задаёт максимальное количество потоков).

```
omp_set_num_threads(num_of_threads)
```

Листинг 7 – функция omp_set_num_threads

Функция из листинга 7 позволяет установить используемое в параллельных областях кода количество потоков (для тех областей, где число потоков не указано явно).

```
omp_get_max_threads()
```

Листинг 8 – функция omp_get_max_threads

Функция из листинга 8 позволяет получить максимальное число тредов, которое использовалось в параллельных регионах кода, где число используемых тредов не было указано явно (то есть там использовалось не более чем столько тредов).

```
omp_get_wtime()
```

Листинг 9 – функция omp_get_wtime

Функция из листинга 9 позволяет получить время, прошедшее с «некоторого момента в прошлом» (эта величина произвольная, но гарантированно не меняется в течении работы программы).

Описание работы написанного кода.

Сначала происходит проверка аргументов командной строки и считывание входного файла в try-catch блоке (листинг 10). Также здесь устанавливается количество тредов (функции из листингов 6-7) (на число тредов установлено ограничение: не меньше -1 и не больше 2000).

```
if (num_of_args != 4) {
    std::cout << "4 arguments expected, found " << num_of_args;
    return 0;
}
int num_of_threads;
try {
    num_of_threads = std::atoi(args[1]);
    if (num_of_threads > 0) {
        omp_set_num_threads(num_of_threads);
        omp_set_dynamic(num_of_threads);
    }
    if (num_of_threads < -1 || num_of_threads > 2000) {
        std::cout << "Illegal number of threads: " << num_of_threads;
        return 0;
    }
} catch (...) {
    std::cout << "Illegal number of threads: " << args[1];
    return 0;
}
std::string in_file = args[2];
std::string out_file = args[3];
std::ifstream in;
char *image, *file;
int image_size, width, height;
try {
    in.open(in_file);

    // getting length of file
    in.seekg(0, std::ios::end);
    long long length = in.tellg();
    in.seekg(0, std::ios::beg);

    // reading file
    file = new char[length];
    in.read(file, length);

    // reading supporting info
    in.seekg(0, std::ios::beg);
    std::string p5;
    in >> p5;
    int n;
    in >> width >> height >> n;
    image_size = width * height;
    image = file + length  - image_size;
    in.close();

    if (p5 != "P5") {
        std::cout << "Not PNM (P5) file";
        return 0;
    }
    if (n != 255) {
        std::cout << "Illegal file format";
```

```
        return 0;
    }

} catch (std::exception const &e) {
    std::cout << "Cannot open/read input file: " << e.what() << '\n';
    return 0;
}
```

Листинг 10 – проверка аргументов командной строки и считывание входного файла

После этого начинается отсчёт времени (функция из листинга 9).

Далее мы строим диаграмму входного изображения (листинг 11). Здесь и далее у каждого параллельного блока кода будет использовано if (num_of_threads != -1), так как -1 соответствует запуску без openmp. Для каждого треда создается отдельный массив гистограммы, в распараллеленом цикле for он заполняется, после чего в блоке critical все такие массивы сливаются в один.

```
//#define SCHEDULE static
// histogram calculation
    int bar_graph[NUM_OF_COLORS];
    std::fill(bar_graph, bar_graph + NUM_OF_COLORS, 0);
#pragma omp parallel if (num_of_threads != -1)
    {
        int bar_graph_th[NUM_OF_COLORS];
        std::fill(bar_graph_th, bar_graph_th + NUM_OF_COLORS, 0);
#pragma omp for nowait schedule(SCHEDULE)
        for (int i = 0; i < image_size; ++i) {
            bar_graph_th[(unsigned char) image[i]]++;
        }
#pragma omp critical
        {
            for (int i = 0; i < NUM_OF_COLORS; ++i) {
                bar_graph[i] += bar_graph_th[i];
            }
        }
    }
```

Листинг 11 – построение диаграммы изображения

Опишем, по какой функции подбираются оптимальные пороги.

$$p(f) - \text{вероятность яркости } f,$$

$$p(f) = \frac{n_f}{N} (n_f - \text{число пикселей яркости } f,$$

$$N - \text{число пикселей в картинке})$$

$$M = 4 - \text{число кластеров}$$

$$q_k - \text{вероятность кластера } k$$

$$\mu_k = \sum_{f \in \text{кластеру } k} \frac{fp(f)}{q_k} - \text{среднее значение кластера } k$$

Тогда искомое значение равно

$$argmax \sum_{k=1}^{M} q_k \mu_k^2 = argmax \sum_{k=1}^{M} q_k \left(\sum_{f \in \text{кластеру } k} \frac{fp(f)}{q_k} \right)^2$$

$$= argmax \sum_{k=1}^{M} \frac{1}{q_k} \left(\sum_{f \in \text{кластеру } k} fp(f) \right)^2$$

Далее считаются два массива префиксных сумм (сумма вероятностей для каждого цвета и сумма матожиданий каждого цвета) (их вычисление не распараллелено, так как цикл всегда делает ровно 256 итераций (то есть не очень много) (листинг 12). Они будут использованы для вычисления дисперсии при подборе оптимального набора порогов.

```
// supporting info calculation
double probability[NUM_OF_COLORS + 1];
double math_expects[NUM_OF_COLORS + 1];
probability[0] = 0;
math_expects[0] = 0;
for (int i = 0; i < NUM_OF_COLORS; ++i) {
    probability[i + 1] = probability[i] + bar_graph[i];
    math_expects[i + 1] = math_expects[i] + bar_graph[i] * i;
}
```

Листинг 12 – подсчёт префиксных сумм

Далее перебираются все возможные варианты порогов (листинг 13). Распараллелен последний цикл, так как по результатам тестирования именно это даёт наилучшее время. Если распараллелить первый цикл, то при static распределении итераций треды будут работать разное количество времени, то есть одни завершатся раньше других, а при dynamic распределении итераций будет совершаться длительная по времени раздача блоков. В нашем случае время будет тратиться на создание тредов на каждой итерации второго цикла, но, как показывает тестирование, это оказывается быстрее. Распараллеливание организовано так же, как в коде в листинге 11: для каждого треда создается локальная переменная максимума, далее в critical-блоке ищется глобальный максимум.

```
// searching for the best combination of threshold
    triple<int> best_combination;
    double max_dispersion = -1;
#pragma omp parallel if (num_of_threads != -1)
    {
```

```
        triple<int> best_combination_th;
        double max_dispersion_th = -1;
        for (int f1 = 0; f1 < NUM_OF_COLORS - 3; ++f1) {
            for (int f2 = f1 + 1; f2 < NUM_OF_COLORS - 2; ++f2) {
#pragma omp for nowait schedule(SCHEDULE)
                for (int f3 = f2 + 1; f3 < NUM_OF_COLORS - 1; ++f3) {
                    double dispersion = math_expects[f1 + 1] * math_expects[f1 +
1] / probability[f1 + 1] +
                                        (math_expects[f2 + 1] - math_expects[f1 +
1]) *
                                        (math_expects[f2 + 1] - math_expects[f1 +
1]) / (probability[f2 + 1] - probability[f1 + 1]) +
                                        (math_expects[f3 + 1] - math_expects[f2 +
1]) *
                                        (math_expects[f3 + 1] - math_expects[f2 +
1]) / (probability[f3 + 1] - probability[f2 + 1]) +
                                        (math_expects[NUM_OF_COLORS] -
math_expects[f3 + 1]) *
                                        (math_expects[NUM_OF_COLORS] -
math_expects[f3 + 1]) / (probability[NUM_OF_COLORS] - probability[f3 + 1]);
                    if (dispersion > max_dispersion_th) {
                        max_dispersion_th = dispersion;
                        best_combination_th = {f1, f2, f3};
                    }
                }
            }
        }
#pragma omp critical
        {
            if (max_dispersion_th > max_dispersion) {
                max_dispersion = max_dispersion_th;
                best_combination = best_combination_th;
            }
        }
    }
```

Листинг 13 – перебор пороговых значений

Далее строится новая картинка. Цикл также распараллелен; разные треды пишут в разные ячейки массива image (листинг 14).

```
// building new image
#pragma omp parallel for if (num_of_threads != -1) schedule(SCHEDULE)
    for (int i = 0; i < image_size; ++i) {
        int pixel = (unsigned char) image[i];
        if (pixel <= best_combination.first) {
            image[i] = FIRST_CLUSTER_VALUE;
        } else if (pixel <= best_combination.second) {
            image[i] = SECOND_CLUSTER_VALUE;
        } else if (pixel <= best_combination.third) {
            image[i] = THIRD_CLUSTER_VALUE;
        } else {
            image[i] = FOURTH_CLUSTER_VALUE;
        }
    }
```

Листинг 14 – построение нового изображения

Далее заканчивается отсчёт времени (функция из листинга 9), итоговое изображение записывается в выходной файл и выводятся логи (листинг 15). Число тредов равно 0, если код запускался без openmp, иначе оно определяется функцией из листинга 8.

```
std::ofstream out;
try {
    out.open(out_file);
    out << "P5\n" << width << ' ' << height << "\n255\n";
    out.write(image, image_size);
    out.close();
} catch (std::exception const &e) {
    std::cout << "Cannot open/read output file: " << e.what() << '\n';
}

delete[] file;
printf("%u %u %u\n", best_combination.first, best_combination.second,
best_combination.third);
printf("Time (%i thread(s)): %g ms\\n", (num_of_threads == -1) ? 0 :
omp_get_max_threads(), (end - start) * 1000);
```

Листинг 15 – запись полученного изображения в выходной файл и вывод логов

Можно заметить, что компилятору и процессору несложно найти независимые действия (там, где они есть (например, при вычислении формулы для дисперсии или при подсчёте префиксных сумм)) и выполнить их параллельно.

Результат работы написанной программы с указанием процессора, на котором производилось тестирование.

Лог вывода программы:

77 130 187

Time (8 thread(s)): 8.05822 ms\n

Процессор:

Intel Core i7 1165G7

4 ядра, 8 потоков

максимальная тактовая частота 4,70 GHz, кэш 12 MB

Экспериментальная часть.

Во всех случаях время работы программы усредняется по 5000 запусков.

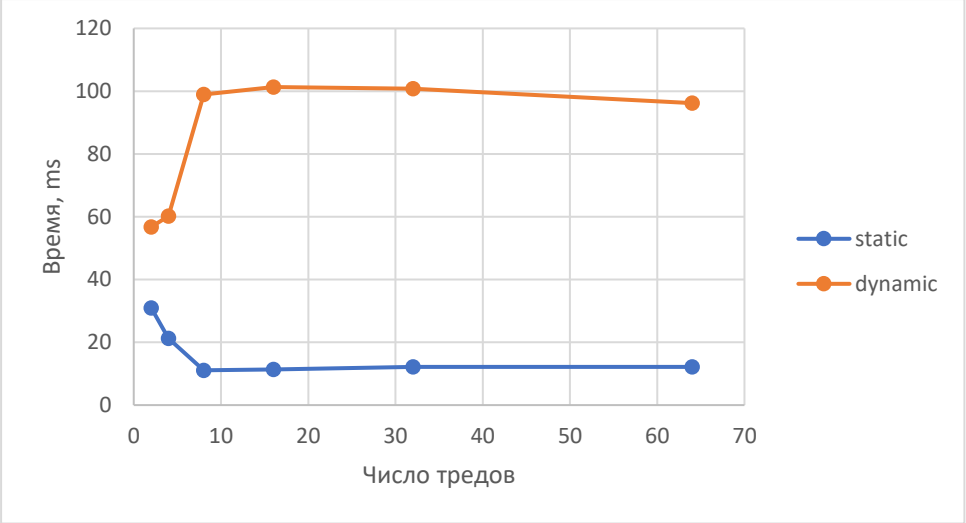

График 1 – зависимость времени работы программы от числа тредов и параметра schedule (без chink_size)

Для динамического распределения итераций между тредами требуется много дополнительных вычислений, поэтому в нашем случае оно сильно проигрывает статическому распределению. По этой же причине время работы программы при динамическом распределении итераций между тредами (поначалу) увеличивается при увеличении числа тредов. При статическом распределении время (поначалу) уменьшается (большее количество итераций выполняются параллельно, а дополнительные вычисления в большом объеме не требуются). Но уже начиная с 8 потоков время фактически выходит на асимптоту и перестает меняться. Это связано с тем, что хоть итерации и выполняются параллельно, не все они выполняются одновременно (ведь число ядер процессора ноутбука (и в целом его вычислительные ресурсы) ограничено).

На графиках 2-7 изображены зависимости времени работы программы от параметров schedule при разном числе потоков.

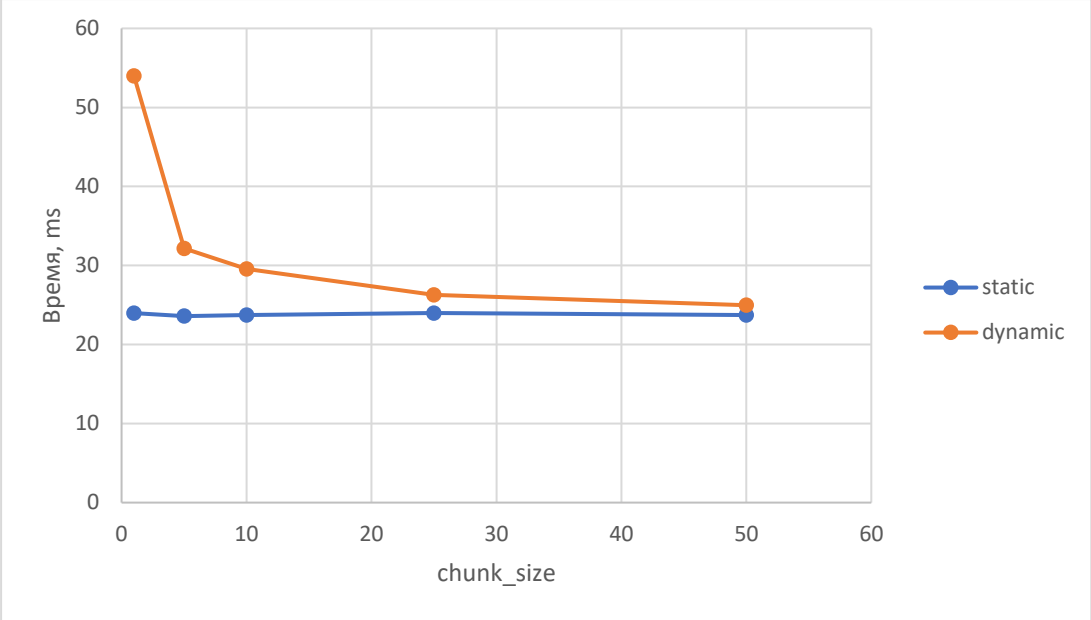

График 2 – зависимость времени работы программы от параметров schedule (2 потока)

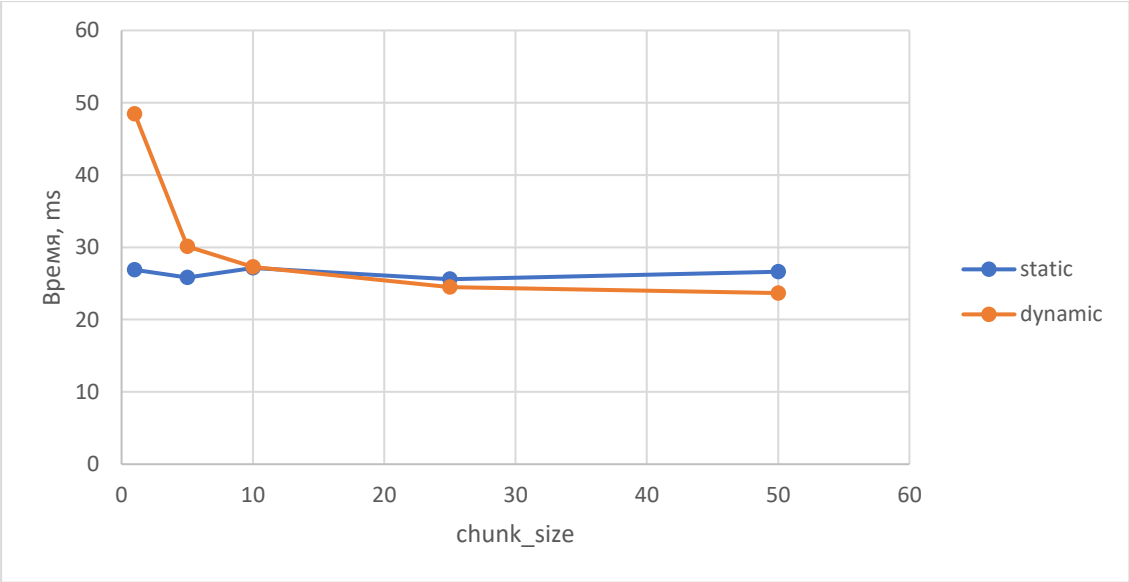

График 3 – зависимость времени работы программы от параметров schedule (4 потока)

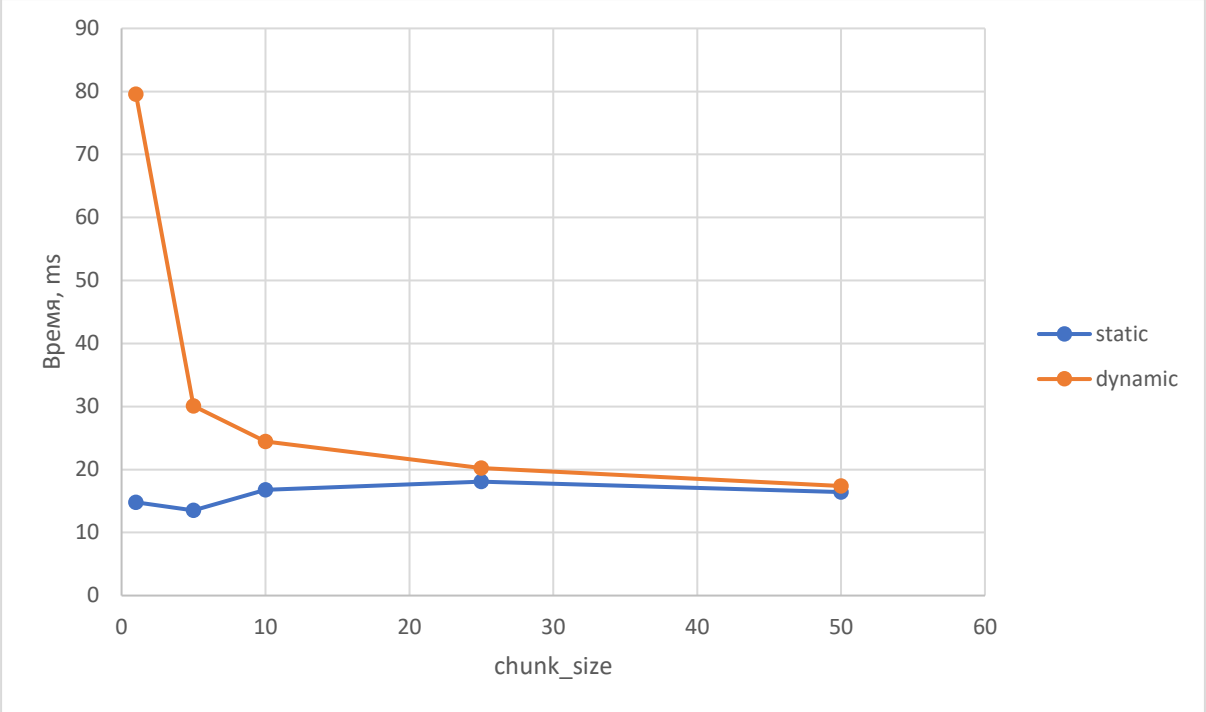

График 4 – зависимость времени работы программы от параметров schedule (8 потоков)

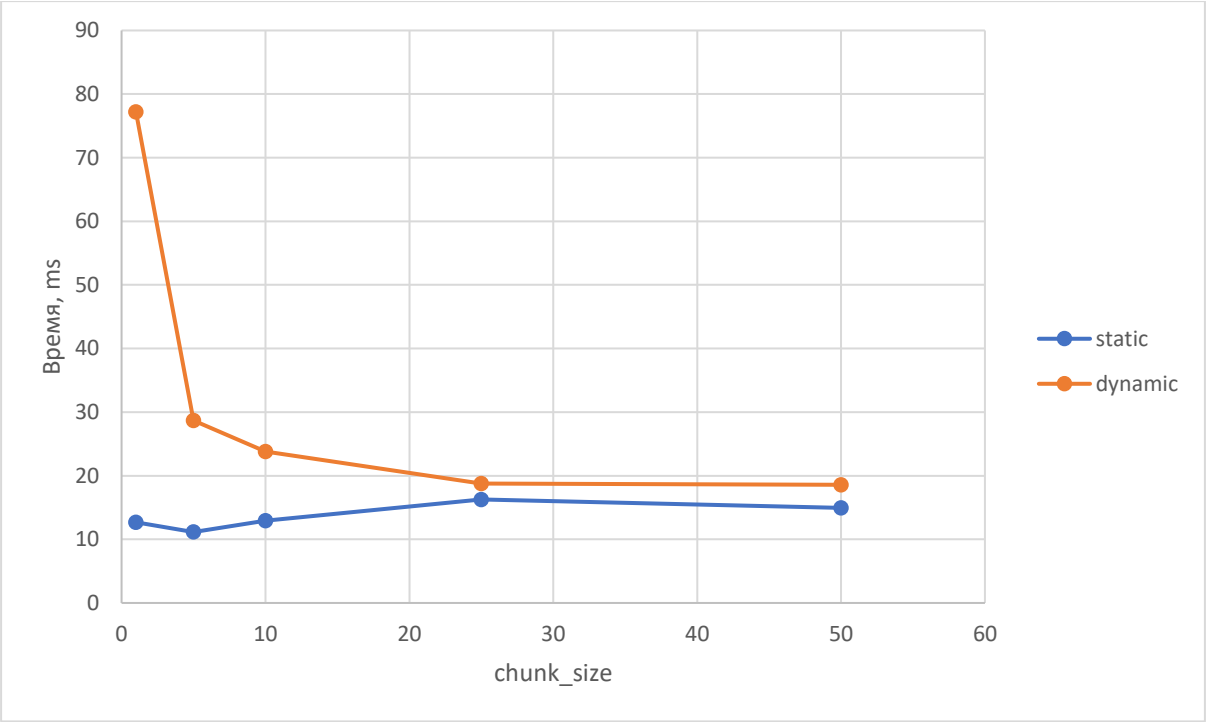

График 5 – зависимость времени работы программы от параметров schedule (16 потоков)

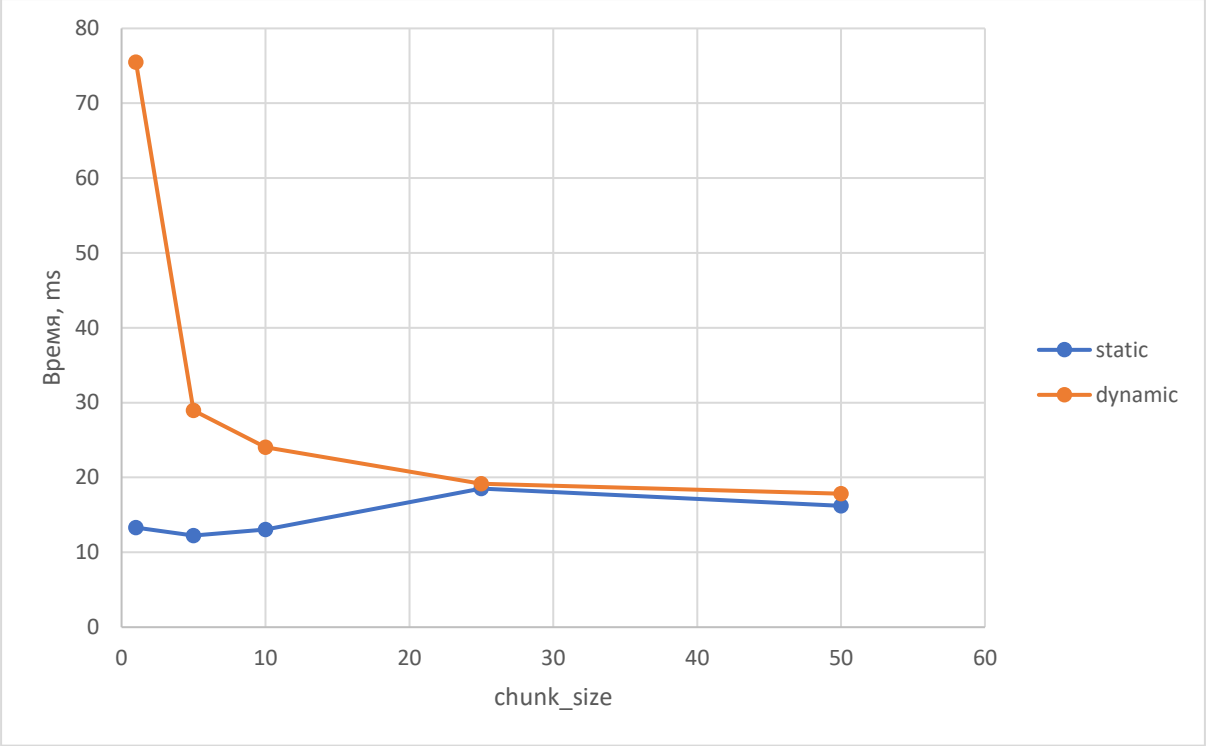

График 6 – зависимость времени работы программы от параметров schedule (32 потока)

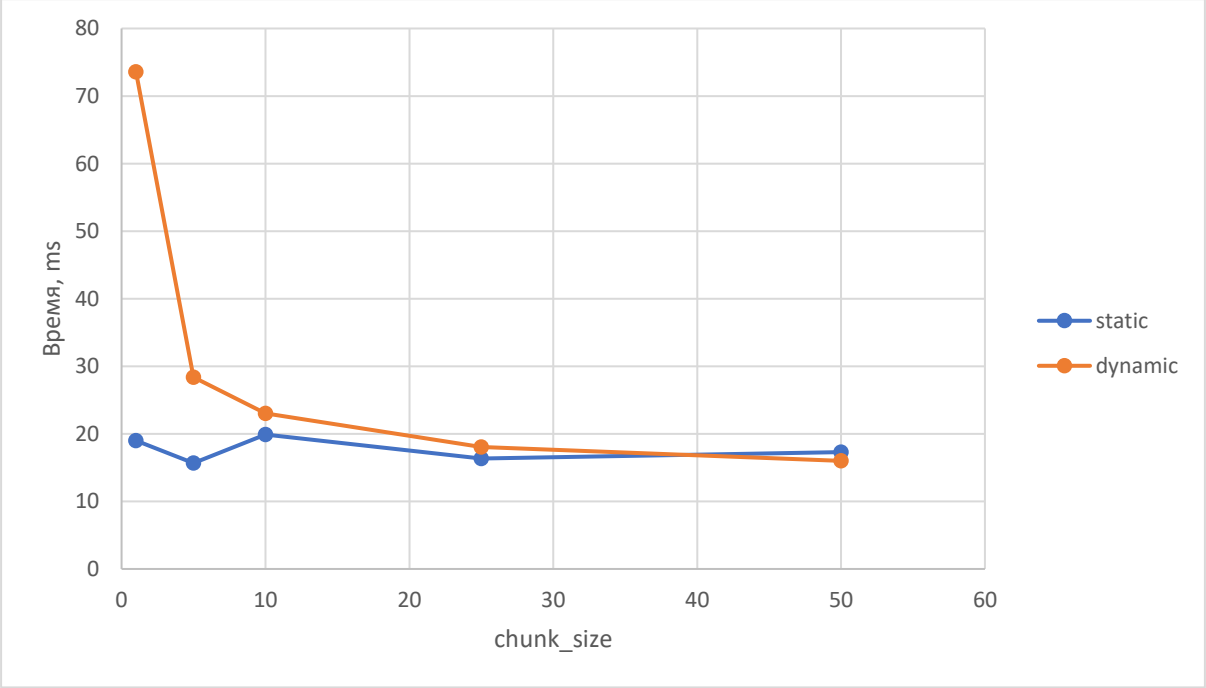

График 7 – зависимость времени работы программы от параметров schedule (64 потока)

На всех графиках видна одна и та же закономерность: для dynamic резкое падение в начале и выход на асимптоту, для static небольшое падение времени работы при chunk_size = 5, после выход на асимптоту (причём

графики довольно близки друг к другу). Резкое падение для dynamic при увеличении chunk_size объясняется более редким перераспределением блоков итераций между тредами, и, как следствие, более редкой необходимость совершать длительные по времени пересчёты. При увеличении chunk_size из-за величины блоков итераций, во-первых, static и dynamic режимы становятся довольно близки по времени (в dynamic режиме перераспределение блоков происходит всё реже и реже), во-вторых, время работы программы выходит на асимптоту (для static режима оно и так менялось не очень сильно, а причины приближения графика dynamic режима к последнему были описаны ранее).

Значения chunk_size, большие 50, не рассматривались при тестировании, так как тестовые входные данные не очень большие, а цикл перебора порогов имеет порядка 250 итераций, то есть если ещё повысить chunk_size, некоторые из потоков не будут использоваться.

По итогам экспериментальной части было решено выбрать static режим распределения итераций между тредами и оставить chunk_size равным значению по умолчанию (результаты замеров времени работы кода без chunk_size и с chunk_size = 5 были очень близки, но, всё же, значение по умолчанию выигрывает при большом числе тредов и не очень сильно проигрывает при маленьком числе тредов).

После этого было произведено тестирование кода с выключенным openmp и с включенным с 1 потоком. Итоги можно увидеть в таблице 1.

1 тред	30.884
Без openmp	28.124

Таблица 1 – результаты тестирования кода с выключенным openmp и с включенным с 1 потоком

С одним тредом код отработал немного дольше, так как openmp совершает дополнительные расчёты при распараллеливании даже на 1 тред (некоторые из них описывались выше). Но время работы отличается не очень сильно, что свидетельствует о том, что производимые дополнительные расчёты не слишком затратны по времени.

Список источников.

В работе использовалась только документация OpenMP API (version 2.0): https://www.openmp.org//wp-content/uploads/cspec20.pdf

Листинг кода.

`hard.cpp`

```cpp
#include <omp.h>
#include <fstream>
#include <iostream>
#include <algorithm>
#include <exception>
#include <cstdlib>
#include <string>
#include <ios>


#define SCHEDULE static

int NUM_OF_COLORS = 256;
char FIRST_CLUSTER_VALUE = 0;
char SECOND_CLUSTER_VALUE = 84;
char THIRD_CLUSTER_VALUE = -86;
char FOURTH_CLUSTER_VALUE = -1;

template<typename T>
struct triple {
    T first, second, third;
};

int main(int num_of_args, char *args[]) {
    if (num_of_args != 4) {
        std::cout << "4 arguments expected, found " << num_of_args;
        return 0;
    }
    int num_of_threads;
    try {
        num_of_threads = std::atoi(args[1]);
        if (num_of_threads > 0) {
            omp_set_num_threads(num_of_threads);
            omp_set_dynamic(num_of_threads);
        }
        if (num_of_threads < -1 || num_of_threads > 2000) {
            std::cout << "Illegal number of threads: " << num_of_threads;
            return 0;
        }
    } catch (...) {
        std::cout << "Illegal number of threads: " << args[1];
        return 0;
    }
    std::string in_file = args[2];
    std::string out_file = args[3];
    std::ifstream in;
    char *image, *file;
    int image_size, width, height;
    try {
        in.open(in_file);

        // getting length of file
        in.seekg(0, std::ios::end);
        long long length = in.tellg();
        in.seekg(0, std::ios::beg);

        // reading file
        file = new char[length];
        in.read(file, length);
```

```
        // reading supporting info
        in.seekg(0, std::ios::beg);
        std::string p5;
        in >> p5;
        int n;
        in >> width >> height >> n;
        image_size = width * height;
        image = file + length  - image_size;
        in.close();

        if (p5 != "P5") {
            std::cout << "Not PNM (P5) file";
            return 0;
        }
        if (n != 255) {
            std::cout << "Illegal file format";
            return 0;
        }

    } catch (std::exception const &e) {
        std::cout << "Cannot open/read input file: " << e.what() << '\n';
        return 0;
    }

    double start = omp_get_wtime();

    // histogram calculation
    int bar_graph[NUM_OF_COLORS];
    std::fill(bar_graph, bar_graph + NUM_OF_COLORS, 0);
#pragma omp parallel if (num_of_threads != -1)
    {
        int bar_graph_th[NUM_OF_COLORS];
        std::fill(bar_graph_th, bar_graph_th + NUM_OF_COLORS, 0);
#pragma omp for nowait schedule(SCHEDULE)
        for (int i = 0; i < image_size; ++i) {
            bar_graph_th[(unsigned char) image[i]]++;
        }
#pragma omp critical
        {
            for (int i = 0; i < NUM_OF_COLORS; ++i) {
                bar_graph[i] += bar_graph_th[i];
            }
        }
    }

    // supporting info calculation
    double probability[NUM_OF_COLORS + 1];
    double math_expects[NUM_OF_COLORS + 1];
    probability[0] = 0;
    math_expects[0] = 0;
    for (int i = 0; i < NUM_OF_COLORS; ++i) {
        probability[i + 1] = probability[i] + bar_graph[i];
        math_expects[i + 1] = math_expects[i] + bar_graph[i] * i;
    }

    // searching for the best combination of threshold
    triple<int> best_combination;
    double max_dispersion = -1;
```

```
#pragma omp parallel if (num_of_threads != -1)
    {
        triple<int> best_combination_th;
        double max_dispersion_th = -1;
        for (int f1 = 0; f1 < NUM_OF_COLORS - 3; ++f1) {
            for (int f2 = f1 + 1; f2 < NUM_OF_COLORS - 2; ++f2) {
#pragma omp for nowait schedule(SCHEDULE)
                for (int f3 = f2 + 1; f3 < NUM_OF_COLORS - 1; ++f3) {
                    double dispersion = math_expects[f1 + 1] * math_expects[f1 +
1] / probability[f1 + 1] +
                                        (math_expects[f2 + 1] - math_expects[f1 +
1]) *
                                        (math_expects[f2 + 1] - math_expects[f1 +
1]) / (probability[f2 + 1] - probability[f1 + 1]) +
                                        (math_expects[f3 + 1] - math_expects[f2 +
1]) *
                                        (math_expects[f3 + 1] - math_expects[f2 +
1]) / (probability[f3 + 1] - probability[f2 + 1]) +
                                        (math_expects[NUM_OF_COLORS] -
math_expects[f3 + 1]) *
                                        (math_expects[NUM_OF_COLORS] -
math_expects[f3 + 1]) / (probability[NUM_OF_COLORS] - probability[f3 + 1]);
                    if (dispersion > max_dispersion_th) {
                        max_dispersion_th = dispersion;
                        best_combination_th = {f1, f2, f3};
                    }
                }
            }
        }
#pragma omp critical
        {
            if (max_dispersion_th > max_dispersion) {
                max_dispersion = max_dispersion_th;
                best_combination = best_combination_th;
            }
        }
    }

    // building new image
#pragma omp parallel for if (num_of_threads != -1) schedule(SCHEDULE)
    for (int i = 0; i < image_size; ++i) {
        int pixel = (unsigned char) image[i];
        if (pixel <= best_combination.first) {
            image[i] = FIRST_CLUSTER_VALUE;
        } else if (pixel <= best_combination.second) {
            image[i] = SECOND_CLUSTER_VALUE;
        } else if (pixel <= best_combination.third) {
            image[i] = THIRD_CLUSTER_VALUE;
        } else {
            image[i] = FOURTH_CLUSTER_VALUE;
        }
    }

    double end = omp_get_wtime();

    std::ofstream out;
    try {
        out.open(out_file);
        out << "P5\n" << width << ' ' << height << "\n255\n";
```

```
        out.write(image, image_size);
        out.close();
    } catch (std::exception const &e) {
        std::cout << "Cannot open/read output file: " << e.what() << '\n';
    }

    delete[] file;
    printf("%u %u %u\n", best_combination.first, best_combination.second,
best_combination.third);
    printf("Time (%i thread(s)): %g ms\\n", (num_of_threads == -1) ? 0 :
omp_get_max_threads(), (end - start) * 1000);

    return 0;
}
```