| Лабораторная работа №4 | M3136 | 2023 |
|---|---|---|
| OpenMP | Ястребов Григорий Дмитриевич | |

**Цель работы:** знакомство с основами многопоточного программирования.

**Инструментарий и требования к работе:** код написан на языке C стандарта C99, в качестве компилятора использовался GCC 11.2.0, стандарт OpenMP 2.0.

**Описание**

Написать программу, решающую поставленную задачу, на языке C/C++, используя OpenMP для распараллеливания вычислений. Провести серию экспериментов по измерению времени работы программы в зависимости от значений параметров, указанных в конструкциях OpenMP в коде.

**Вариант:** Hard. Пороговая фильтрация изображения методом Оцу. Необходимо реализовать алгоритм для трёх порогов.

# Описание OpenMP

OpenMP – набор директив, библиотечных функций, и переменных окружения, предназначенных для параллелизации исполнения команд в коде программы C/C++.

При запуске программы с включенным OpenMP код начинает исполняться на “мастер-потоке”, который имеет свойство в соответствующем “блоке параллелизации” делиться на “команду” потоков для одновременного исполнения кода, а затем собираться обратно. В этом блоке каждый поток уже сам является собственным мастер-потоком и может делиться дальше.

Начало блока параллелизации обозначается директивой #pragma omp parallel. Внутри блока параллелизации могут объявляться дополнительные директивы-конструкции, указывающие команде потоков, как следует выполнять последующий код.

## Конструкции

#pragma omp for – распределяет итерации цикла между потоками. Чтобы не возникло ошибки race condition – одновременной записи/чтения несколькими потоками и, соответственно, неопределенному поведению, в теле цикла не должен содержаться код, зависящий от результата выполнения других итераций.

В качестве параметров может принимать schedule(kind, chunk_size) – формат распределения итераций между потоками.

static kind делит все итерации на блоки размера chunk_size и по кругу передает их потокам. По умолчанию значение chunk_size ставится такое,

что на каждый поток приходится примерно одинаковое количество итераций.

dynamic kind делит все итерации на блоки размера chunk_size и динамически выдает их свободным потокам. Получив блок, поток не является свободным, пока не исполнит его полностью. По умолчанию значение chunk_size установлено на 1.

Таким образом, dynamic kind позволяет “сгладить углы” и приблизить время исполнения блока итераций к среднему значению по потокам, что может быть важно, когда разные итерации цикла могут выполняться за разнящееся время.

#pragma omp critical – последующий код выполняется последовательно всеми потоками, одновременно его может выполнять только один поток.

Пример использования директив и конструкций:

```
int pnum[COLORS] = {}; // массив создан мастер-потоком, общий для всех
#pragma omp parallel if (tnum > 0) // разделение на команду потоков
{
   int tpnum[COLORS] = {}; // частный массив для каждого потока
   #pragma omp for     // итерации цикла распределяются между потоками
   for (int i = 0; i < MSIZE; ++i) {
       ++tpnum[img->matr[i]];  // запись в частный массив потока
   }
   #pragma omp critical    // последовательное выполнение потоками
   {
       for (int i = 0; i < COLORS; ++i) { // данные из частного
           pnum[i] += tpnum[i];    // массива сливаются в общий
       }
   }
}
```

## Runtime-функции

double omp_get_wtime() – получить текущее (относительное) время во время исполнения программы (в секундах).

void omp_set_num_threads(int) – установить количество потоков.

int omp_get_max_threads() – получить максимальное доступное количество потоков.

# Описание кода программы

Весь исходный код программы находится в директории src и состоит из файлов hard.c, image.c, filter.c и хэдер-файлов filter.h, image.h.

В hard.c находится функция main. В ней обрабатываются переданные аргументы, устанавливается число потоков tnum через omp_set_num_threads, далее создается экземпляр img структуры Image и записывается из входного файла (read_pnm). Вызывается filter4(img, ..), делаются замеры времени работы через omp_get_wtime. В конце img записывается в выходной файл (write_pnm) и освобождается. Функция также ловит и обрабатывает ошибки.

В image.c содержится описание структуры Image, в которой хранятся данные об изображении – длина и ширина в пикселях, и одномерный массив всех пикселей. Есть конструктор и деструктор (img_init, img_destroy) и функции для чтения и записи файла формата PGM в переданный экземпляр Image (read_pnm, write_pnm). Во всех функциях есть обработка ошибок.

В filter.c функция filter4 принимает экземпляр img структуры Image и tnum, выполняет пороговую фильтрацию по 4 кластерам. Внутри вызываются функции hist (получение гистограммы) и otsu3 (нахождение 3 пороговых значений для кластеров). В этих функциях применяется параллелизация.

## Параллелизация кода

Функция filter4 собирает гистограмму изображения через hist и далее передает ее в otsu3, откуда получает пороговые значения для кластеров. В конце она проходит циклом по всем точкам изображения и

перекрашивает в соответствие с надлежащим кластером. В этом цикле применяется директива #pragma omp parallel for if (tnum > 0), которая выполняется, если через аргументы передано не -1 (здесь и далее в разделе параметр schedule не уточняется). Таким образом, каждой точке изображения достается один поток, который ее перезапишет, race-condition не происходит.

В функцию hist передается массив размером 256 (количество возможных цветов у точки изображения). Цикл for проходит по всем точкам и прибавляет 1 в этот массив по индексу цвета этой данной точки. Чтобы избежать race-condition (обращение к одному индексу у массива гистограммы), for оборачивается в parallel-блок, в котором для каждого потока создается собственный массив гистограммы. Частные массивы также записывается через for, но держат лишь часть гистограммы. Поэтому в конце мы через critical-блок по очереди из каждого потока сливаем частный массив гистограммы в общий (см. пример из описания OpenMP).

Функция otsu3 выполняет алгоритм определения . Здесь считаются массивы вероятностей и матожиданий для каждой точки, а также префиксные суммы этих массивов для быстрого нахождения суммы на отрезке (по двум границам кластера). Поскольку итераций в циклах мало, и 2 из них нельзя выполнить параллельно (для рассчета префиксной суммы массива нужно знать значение в предыдущем индексе), здесь нет parallel-блока. Он начинается далее, когда полным перебором в три вложенных цикла определяются 3 границы кластеров. В теле цикла высчитывается пара десятков double, затем обновляется максимальное значение дисперсии. Здесь снова есть race-condition на обновлении максимума и соответствующих ему порогов, поэтому каждый поток

хранит свой максимум и пороги, а затем они сравниваются между собой в critical-блоке.

## Результат работы программы

Программа запускалась на процессоре Apple M1 (Macbook Air 2020), 8 ядер, 8 потоков.

Входные данные:

```
omp4 0 test_data/in.pgm test_data/out.pgm
```

Выходные данные:

```
77 130 187

Time (8 thread(s)): 12.716 ms
```

## Экспериментальная часть

Был написан bash-скрипт (test_script.sh, лежит в корне репозитория), запускающий программу произвольное количество раз на 1-8 потоках, а также на 1 потоке с включенным OpenMP и выключенным OpenMP. Выводится время 5 последовательных запусков и их среднее (см. пункт “результат работы скрипта-тестера”).

Также в тестовые файлы было добавлено 8k-изображение (test_data/in8k.pgm). При запуске программы на нем основная нагрузка будет приходиться на подсчет гистограммы.

На графиках (см. рисунок 1 и рисунок 2) представлены результаты замеров времени при запуске на 1-8 потоках при следующих наборах параметров:

1. Все schedule kind установлены на static, chunk_base не установлен.
2. Все schedule kind установлены на dynamic, chunk_base не установлен.
3. (dynamic, 256) у циклов, работающих с матрицей изображения, (static, default) у цикла выбора порогов.
4. (dynamic, 4096) у циклов, работающих с матрицей изображения, (dynamic, default) у цикла выбора порогов.

В случае изображения 8K все значения времени у 2 набора превышали 600 мс и с увеличением числа потоков лишь только росли.

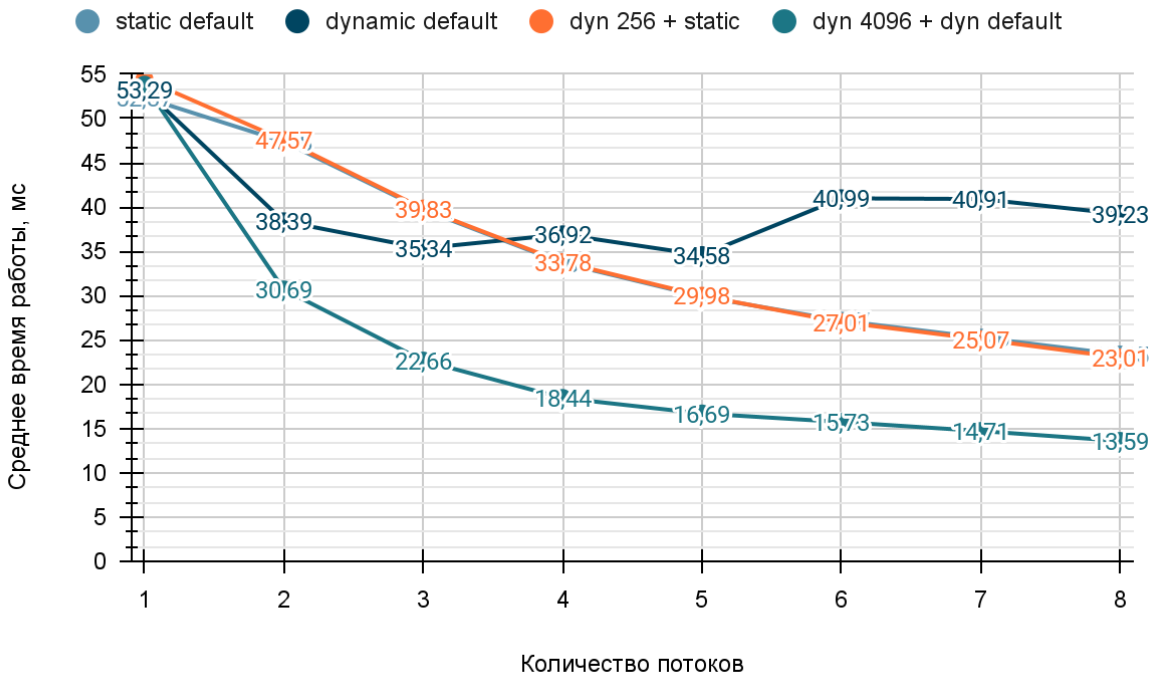


Рисунок 1. График зависимости среднего времени работы программы от количества потоков при разных наборах schedule. Изображение 500x500.

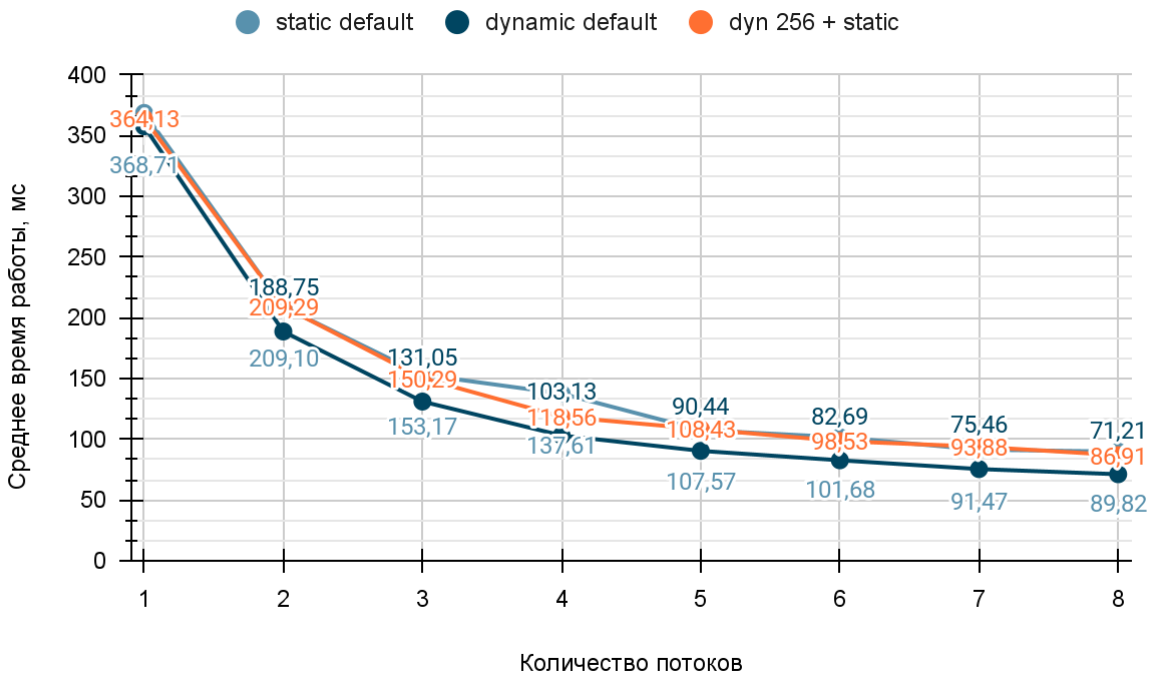


Рисунок 2. График зависимости среднего времени работы программы от количества потоков при разных наборах schedule. Изображение 8K.

Как и ожидалось, с увеличением числа задействованных потоков время работы программы будет уменьшаться, в среднем, около пропорционально. Замеры показывают, что при правильном выборе chunk_base у dinamic kind for может работать быстрее, чем static kind. Особенно ощутим прирост в случае изображения 500x500, где основная нагрузка лежит на выборе порогового значения. Каждая итерация цикла довольно тяжеловесная и может разниться по времени с остальными, поэтому с dynamic kind цикл выполняется быстрее.

Время работы программы с включенным OpenMP на одном потоке: 46.75 мс для изображения 500x500 и 367.50 мс для 8k-изображения. Время работы программы с выключенным OpenMP: 47.03 мс и 352.15 мс соответственно. Существенного прироста времени работы нет.

## Результат работы скрипта-тестера

**schedule(static, default)**

```
Testing parallel execution with OMP
Test 1: 5 test cases, 0.50 seconds interval
input: test_data/in.pgm, output: test_data/out.pgm
1 threads: 74.61 ms | 47.53 ms | 46.56 ms | 46.54 ms | 46.60 ms | avg 52.37 ms
2 threads: 74.19 ms | 42.26 ms | 40.19 ms | 40.14 ms | 40.11 ms | avg 47.38 ms
3 threads: 65.15 ms | 35.25 ms | 32.70 ms | 32.84 ms | 32.65 ms | avg 39.72 ms
4 threads: 55.48 ms | 31.60 ms | 27.14 ms | 26.98 ms | 27.21 ms | avg 33.68 ms
5 threads: 49.19 ms | 29.10 ms | 23.89 ms | 23.54 ms | 23.75 ms | avg 29.89 ms
6 threads: 44.67 ms | 27.00 ms | 21.21 ms | 21.47 ms | 21.52 ms | avg 27.17 ms
7 threads: 41.46 ms | 24.78 ms | 21.64 ms | 19.85 ms | 18.57 ms | avg 25.26 ms
8 threads: 37.21 ms | 23.77 ms | 19.34 ms | 17.03 ms | 19.41 ms | avg 23.35 ms
Test 2: 5 test cases, 0.50 seconds interval
input: test_data/in8k.pgm, output: test_data/out8k.pgm
1 threads: 389.30 ms | 356.16 ms | 359.21 ms | 357.68 ms | 358.29 ms | avg 364.13
ms
2 threads: 232.47 ms | 204.22 ms | 203.34 ms | 203.34 ms | 203.11 ms | avg 209.29
ms
3 threads: 171.04 ms | 145.32 ms | 145.29 ms | 144.92 ms | 144.87 ms | avg 150.29
ms
4 threads: 139.75 ms | 113.15 ms | 114.04 ms | 112.74 ms | 113.14 ms | avg 118.56
ms
5 threads: 126.87 ms | 106.83 ms | 104.86 ms | 102.76 ms | 100.84 ms | avg 108.43
ms
6 threads: 112.21 ms | 97.22 ms | 94.17 ms | 94.94 ms | 94.11 ms | avg 98.53 ms
7 threads: 115.02 ms | 93.19 ms | 85.11 ms | 85.13 ms | 90.94 ms | avg 93.88 ms
8 threads: 98.43 ms | 81.53 ms | 83.82 ms | 88.38 ms | 82.39 ms | avg 86.91 ms
Testing with OMP off
```

47.41 ms | 47.05 ms | 47.64 ms | 47.32 ms | 47.35 ms | avg 47.35 ms
360.18 ms | 360.93 ms | 360.76 ms | 358.91 ms | 357.90 ms | avg 359.73 ms
Testing with OMP 1 thread
46.83 ms | 47.01 ms | 47.20 ms | 47.43 ms | 47.50 ms | avg 47.19 ms
359.29 ms | 360.13 ms | 359.16 ms | 386.88 ms | 374.73 ms | avg 368.04 ms

**schedule(dynamic, default)**

Testing parallel execution with OMP
Test 1: 5 test cases, 0.50 seconds interval
input: test_data/in.pgm, output: test_data/out.pgm
1 threads: 70.84 ms | 49.32 ms | 48.66 ms | 48.82 ms | 48.79 ms | avg 53.29 ms
2 threads: 62.71 ms | 34.45 ms | 31.82 ms | 31.42 ms | 31.56 ms | avg 38.39 ms
3 threads: 57.71 ms | 32.10 ms | 28.73 ms | 29.10 ms | 29.06 ms | avg 35.34 ms
4 threads: 63.08 ms | 32.77 ms | 29.09 ms | 29.65 ms | 30.01 ms | avg 36.92 ms
5 threads: 34.52 ms | 36.23 ms | 32.10 ms | 35.23 ms | 34.81 ms | avg 34.58 ms
6 threads: 63.76 ms | 39.26 ms | 35.05 ms | 34.28 ms | 32.61 ms | avg 40.99 ms
7 threads: 62.21 ms | 39.08 ms | 34.77 ms | 33.76 ms | 34.75 ms | avg 40.91 ms
8 threads: 50.46 ms | 39.30 ms | 38.13 ms | 32.70 ms | 35.54 ms | avg 39.23 ms
Test 2: 5 test cases, 0.50 seconds interval
input: test_data/in8k.pgm, output: test_data/out8k.pgm
1 threads: 676.12 ms | 648.28 ms | 665.37 ms | 645.96 ms | 648.07 ms | avg 656.76 ms
2 threads: 1169.08 ms | 1120.35 ms | 1125.08 ms | 1121.81 ms | 1110.62 ms | avg 1129.39 ms
3 threads: 1735.23 ms | 1684.58 ms | 1680.98 ms | 1657.51 ms | 1666.64 ms | avg 1684.99 ms
4 threads: 2240.00 ms | 2172.71 ms | 2163.98 ms | 2171.00 ms | 2160.89 ms | avg 2181.72 ms
5 threads: 2518.74 ms | 3475.23 ms | 3493.10 ms | 3537.24 ms | 3546.49 ms | avg 3314.16 ms
6 threads: 3106.34 ms | 2985.28 ms | 3278.94 ms | 3302.47 ms | 3327.91 ms | avg 3200.19 ms
7 threads: 3534.75 ms | 3341.16 ms | 3220.07 ms | 3145.04 ms | 3308.21 ms | avg 3309.85 ms
8 threads: 3848.16 ms | 3762.36 ms | 3811.90 ms | 3816.46 ms | 3797.35 ms | avg 3807.25 ms
Testing with OMP off
49.71 ms | 49.76 ms | 49.73 ms | 53.07 ms | 50.76 ms | avg 50.60 ms
653.58 ms | 654.38 ms | 658.26 ms | 655.62 ms | 650.90 ms | avg 654.55 ms
Testing with OMP 1 thread
49.58 ms | 49.57 ms | 49.56 ms | 49.59 ms | 49.55 ms | avg 49.57 ms
648.09 ms | 647.30 ms | 655.03 ms | 653.43 ms | 650.69 ms | avg 650.91 ms

**images schedule(dynamic, 256) + threshold schedule(static, default)**

Testing parallel execution with OMP
Test 1: 5 test cases, 0.50 seconds interval
input: test_data/in.pgm, output: test_data/out.pgm
1 threads: 83.89 ms | 47.85 ms | 46.59 ms | 46.65 ms | 47.15 ms | avg 54.43 ms
2 threads: 74.51 ms | 42.24 ms | 40.33 ms | 40.23 ms | 40.55 ms | avg 47.57 ms
3 threads: 64.66 ms | 36.07 ms | 32.76 ms | 32.65 ms | 33.01 ms | avg 39.83 ms
4 threads: 56.13 ms | 31.41 ms | 27.16 ms | 27.11 ms | 27.08 ms | avg 33.78 ms
5 threads: 50.26 ms | 28.83 ms | 23.75 ms | 23.52 ms | 23.50 ms | avg 29.98 ms
6 threads: 45.56 ms | 26.71 ms | 22.13 ms | 20.35 ms | 20.31 ms | avg 27.01 ms
7 threads: 41.66 ms | 24.63 ms | 20.18 ms | 18.75 ms | 20.11 ms | avg 25.07 ms
8 threads: 37.34 ms | 24.09 ms | 18.78 ms | 18.61 ms | 16.23 ms | avg 23.01 ms
Test 2: 5 test cases, 0.50 seconds interval
input: test_data/in8k.pgm, output: test_data/out8k.pgm

```
1 threads: 385.71 ms | 357.98 ms | 354.62 ms | 391.03 ms | 354.20 ms | avg 368.71
ms
2 threads: 234.73 ms | 202.16 ms | 203.95 ms | 202.35 ms | 202.29 ms | avg 209.10
ms
3 threads: 176.03 ms | 152.22 ms | 146.43 ms | 145.70 ms | 145.46 ms | avg 153.17
ms
4 threads: 171.16 ms | 149.54 ms | 136.08 ms | 118.01 ms | 113.25 ms | avg 137.61
ms
5 threads: 131.58 ms | 101.62 ms | 101.71 ms | 101.54 ms | 101.41 ms | avg 107.57
ms
6 threads: 131.29 ms | 99.09 ms | 92.85 ms | 92.86 ms | 92.30 ms | avg 101.68 ms
7 threads: 112.29 ms | 85.38 ms | 87.70 ms | 85.72 ms | 86.26 ms | avg 91.47 ms
8 threads: 111.97 ms | 84.57 ms | 83.09 ms | 87.71 ms | 81.79 ms | avg 89.82 ms
Testing with OMP off
48.04 ms | 48.77 ms | 47.71 ms | 48.45 ms | 48.19 ms | avg 48.23 ms
369.75 ms | 360.70 ms | 358.42 ms | 359.90 ms | 358.56 ms | avg 361.46 ms
Testing with OMP 1 thread
47.35 ms | 47.25 ms | 49.17 ms | 50.82 ms | 58.86 ms | avg 50.69 ms
381.75 ms | 363.19 ms | 358.95 ms | 358.64 ms | 359.02 ms | avg 364.31 ms
```

**images schedule(dynamic, 4096) + threshold schedule(dynamic, default)**

```
Testing parallel execution with OMP
Test 1: 5 test cases, 0.50 seconds interval
input: test_data/in.pgm, output: test_data/out.pgm
1 threads: 81.61 ms | 47.66 ms | 46.52 ms | 46.53 ms | 46.77 ms | avg 53.82 ms
2 threads: 51.33 ms | 29.23 ms | 24.39 ms | 24.17 ms | 24.31 ms | avg 30.69 ms
3 threads: 38.54 ms | 23.18 ms | 18.09 ms | 16.80 ms | 16.70 ms | avg 22.66 ms
4 threads: 31.16 ms | 19.75 ms | 15.36 ms | 13.12 ms | 12.80 ms | avg 18.44 ms
5 threads: 26.26 ms | 18.13 ms | 14.72 ms | 12.73 ms | 11.61 ms | avg 16.69 ms
6 threads: 24.14 ms | 17.61 ms | 13.86 ms | 12.25 ms | 10.80 ms | avg 15.73 ms
7 threads: 21.09 ms | 16.65 ms | 13.35 ms | 11.66 ms | 10.82 ms | avg 14.71 ms
8 threads: 18.56 ms | 15.02 ms | 12.96 ms | 11.18 ms | 10.21 ms | avg 13.59 ms
Test 2: 5 test cases, 0.50 seconds interval
input: test_data/in8k.pgm, output: test_data/out8k.pgm
1 threads: 378.54 ms | 351.79 ms | 353.83 ms | 352.82 ms | 351.64 ms | avg 357.72
ms
2 threads: 214.86 ms | 182.01 ms | 181.43 ms | 183.46 ms | 182.02 ms | avg 188.75
ms
3 threads: 151.92 ms | 125.65 ms | 125.72 ms | 125.96 ms | 125.99 ms | avg 131.05
ms
4 threads: 122.09 ms | 95.29 ms | 95.38 ms | 95.24 ms | 107.66 ms | avg 103.13 ms
5 threads: 109.17 ms | 86.12 ms | 85.67 ms | 85.70 ms | 85.56 ms | avg 90.44 ms
6 threads: 98.46 ms | 78.09 ms | 80.98 ms | 77.93 ms | 77.99 ms | avg 82.69 ms
7 threads: 91.49 ms | 71.61 ms | 71.34 ms | 71.63 ms | 71.25 ms | avg 75.46 ms
8 threads: 83.94 ms | 69.81 ms | 67.10 ms | 68.00 ms | 67.22 ms | avg 71.21 ms
Testing with OMP off
47.35 ms | 47.15 ms | 46.89 ms | 46.72 ms | 47.02 ms | avg 47.03 ms
352.32 ms | 351.32 ms | 351.47 ms | 351.54 ms | 354.09 ms | avg 352.15 ms
Testing with OMP 1 thread
47.00 ms | 46.49 ms | 46.56 ms | 47.01 ms | 46.67 ms | avg 46.75 ms
352.89 ms | 354.97 ms | 357.19 ms | 391.51 ms | 380.92 ms | avg 367.50 ms
```

## Список источников


1. https://www.openmp.org//wp-content/uploads/cspec20.pdf – спецификация OpenMP 2.0
2. https://en.cppreference.com/ – спецификация C
3. https://learn.microsoft.com/en-us/archive/msdn-magazine/2005/october/openmp-and-c-reap-the-benefits-of-multithreading-without-all-the-work – статья об OpenMP на Microsoft Learn

# Листинг кода

**src/hard.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "image.h"
#include "filter.h"

void fail(const char *msg) {
   printf("Application failed with an error: %s\n", msg);
}

int main(int argc, char *argv[]) {

   if (argc != 4){
       fail("invalid arguments");
       printf("expected input: ");
       printf("omp4 <threads count> <PNM input file> <PNM output file>\n");
       return 0;
   }
   char *end;
   int tnum = strtol(argv[1], &end, 10);
   if (argv[1] == end || tnum > omp_get_max_threads()) {
       fail("invalid threads count");
       return 0;
   }
   if (tnum == 0) {
       tnum = omp_get_max_threads();
   }
   omp_set_num_threads(tnum);

   FILE *fin = fopen(argv[2], "rb");
   if (!fin) {
       fail("unable to open input file");
       return 0;
   }
   Image *img = (Image *) malloc(sizeof(Image));
   if (read_pnm(fin, img) == -1) {
       fail("unexpected PNM input file format");
       return 0;
   }
   fclose(fin);

   double tstart = omp_get_wtime();
   filter4(img, tnum);
   double tend = omp_get_wtime();


   FILE *fout = fopen(argv[3], "wb");
   if (!fout) {
       fail("unable to open output file");
       return 0;
   }
   if (write_pnm(fout, img) == -1) {
       fail("unable to write to output file");
   }
```

```
    fclose(fout);
    img_destroy(img);
    free(img);

    printf("Time (%i thread(s)): %g ms\n", tnum, (tend - tstart) * 1000.0);

    return 0;
}
```

**src/image.h**

```
#ifndef IMAGE_H
#define IMAGE_H

#include <stdio.h>
#include <stdint.h>

typedef struct {
    int w;
    int h;
    uint8_t *matr;
} Image;

void img_init(Image *p, int w, int h);

void img_destroy(Image *p);

uint8_t *img_point(Image *p, int i, int j);

int read_pnm(FILE *file, Image *img);

int write_pnm(FILE *file, Image *img);

#endif
```

**src/image.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "image.h"

void img_init(Image *p, int w, int h)
{
    assert(p);
    p->w = w;
    p->h = h;
    p->matr = (uint8_t *) malloc(w * h * sizeof(uint8_t));
}

void img_destroy(Image *p)
{
    assert(p);
    free(p->matr);
}
```

```c
uint8_t *img_point(Image *p, int i, int j)
{
   assert(p);
   return p->matr + (i * p->w + j);
}

int read_pnm(FILE *file, Image *img) {
   assert(file);
   assert(img);
   char format[3];
   if (fscanf(file, "%2s", format) == -1 || strcmp(format, "P5") != 0) {
       return -1;
   }
   int w, h;
   if (fscanf(file, "%i %i", &w, &h) == -1) {
       return -1;
   }
   int m;
   if (fscanf(file, " %i ", &m) == -1 || m != 255) {
       return -1;
   }
   img_init(img, w, h);
   const int dot_count = img->w * img->h;
   if (fread(img->matr, 1, dot_count, file) < dot_count) {
       return -1;
   }
   return 0;
}

int write_pnm(FILE *file, Image *img) {
   assert(file);
   assert(img);
   if (fprintf(file, "P5\n%d %d\n255\n", img->w, img->h) < 0) {
       return -1;
   }
   const int dot_count = img->w * img->h;
   if (fwrite(img->matr, sizeof(uint8_t), dot_count, file) < dot_count) {
       return -1;
   }
   return 0;
}
```

**src/filter.h**

```c
#ifndef FILTER_H
#define FILTER_H

#include "image.h"

void filter4(Image *img, int tnum);

#endif
```

**src/filter.c**

```c
#include <assert.h>
#include <stdlib.h>
```

```c
#include "filter.h"

#define COLORS 256

static void hist(int histo[COLORS], const Image *img, int tnum);

static void otsu3(int thresh[3], const int histo[COLORS], int imgsize, int
tnum);

void filter4(Image *img, const int tnum)
{
   assert(img);
   int histo[COLORS] = {};
   hist(histo, img, tnum);
   int thresh[3];
   otsu3(thresh, histo, img->w * img->h, tnum);
   printf("%u %u %u\n", thresh[0], thresh[1], thresh[2]);


   const uint8_t FILTER[3 + 1] = {0, 84, 170, 255};
   const int MSIZE = img->w * img->h;

   #pragma omp parallel for schedule(dynamic, 4096) if (tnum > 0)
   for (int i = 0; i < MSIZE; ++i)
   {
       uint8_t color = img->matr[i];
       uint8_t fcolor = FILTER[0];
       for (int t = 0; t < 3; ++t)
       {
           if (color > thresh[t])
           {
               fcolor = FILTER[t + 1];
           }
       }
       img->matr[i] = fcolor;
   }
}

static void hist(int histo[COLORS], const Image *img, int tnum)
{
   assert(histo);
   assert(img);
   const int imgsize = img->w * img->h;

   #pragma omp parallel if (tnum > 0)
   {
       int thisto[COLORS] = {}; // hist for each thread
       #pragma omp for schedule (dynamic, 4096)
       for (int i = 0; i < imgsize; ++i) {
           ++thisto[img->matr[i]];
       }
       #pragma omp critical
       {
           for (int i = 0; i < COLORS; ++i) {
               histo[i] += thisto[i];
           }
       }
   }
}
```

```
static void otsu3(int thresh[3], const int histo[COLORS], const int imgsize,
const int tnum)
{
   assert(thresh);
   assert(histo);

   double prob[COLORS];  // brightness probabilities
   for (int i = 0; i < COLORS; ++i) {
       prob[i] = 1.0 * histo[i] / imgsize;
   }
   double prob_prsum[COLORS + 1]; // prob prefix sum
   prob_prsum[0] = 0.0;
   for (int i = 0; i < COLORS; ++i) {
       prob_prsum[i + 1] = prob_prsum[i] + prob[i];
   }

   double exp[COLORS]; // brightness expectations
   for (int i = 0; i < COLORS; ++i) {
       exp[i] = prob[i] * (i + 1);
   }
   double exp_prsum[COLORS + 1]; // exp prefix sum
   exp_prsum[0] = 0.0;
   for (int i = 0; i < COLORS; ++i) {
       exp_prsum[i + 1] = exp_prsum[i] + exp[i];
   }

   double maxdisp = 0.0;
   #pragma omp parallel if (tnum > 0)
   {
       double tmaxdisp = 0.0;
       int tthresh1, tthresh2, tthresh3;
       #pragma omp for schedule(dynamic)
       for (int t1 = 0; t1 < COLORS; ++t1)
       {
           for (int t2 = t1 + 1; t2 < COLORS; ++t2)
           {
               for (int t3 = t2 + 1; t3 < COLORS; ++t3)
               {
                   // cluster brightness probabilities
                   const double clprob1 = prob_prsum[t1 + 1] - prob_prsum[0],
                           clprob2 = prob_prsum[t2 + 1] - prob_prsum[t1 + 1],
                           clprob3 = prob_prsum[t3 + 1] - prob_prsum[t2 + 1],
                            clprob4 = prob_prsum[COLORS - 1] - prob_prsum[t3 +
1];

                   // cluster brightness expectations
                   const double clexp1 = exp_prsum[t1 + 1] - exp_prsum[0],
                           clexp2 = exp_prsum[t2 + 1] - exp_prsum[t1 + 1],
                           clexp3 = exp_prsum[t3 + 1] - exp_prsum[t2 + 1],
                           clexp4 = exp_prsum[COLORS - 1] - exp_prsum[t3 + 1];

                   // average cluster brightnesses
                   const double clavg1 = clprob1 ? clexp1 / clprob1 : 0,
                           clavg2 = clprob2 ? clexp2 / clprob2 : 0,
                           clavg3 = clprob3 ? clexp3 / clprob3 : 0,
                           clavg4 = clprob4 ? clexp4 / clprob4 : 0;

                   // image brightness expectation
                   const double imgexp = clexp1 + clexp2 + clexp3 + clexp4;
```

```
                    const double clexpdiff1 = clavg1 - imgexp,
                            clexpdiff2 = clavg2 - imgexp,
                            clexpdiff3 = clavg3 - imgexp,
                            clexpdiff4 = clavg4 - imgexp;

                    // intercluster dispersion
                    const double disp = clprob1 * clexpdiff1 * clexpdiff1
                                 + clprob2 * clexpdiff2 * clexpdiff2
                                 + clprob3 * clexpdiff3 * clexpdiff3
                                 + clprob4 * clexpdiff4 * clexpdiff4;

                    if (disp > tmaxdisp)
                    {
                        tthresh1 = t1;
                        tthresh2 = t2;
                        tthresh3 = t3;
                        tmaxdisp = disp;
                    }
                }
            }
        }
        #pragma omp critical
        {
            if (tmaxdisp > maxdisp)
            {
                maxdisp = tmaxdisp;
                thresh[0] = tthresh1;
                thresh[1] = tthresh2;
                thresh[2] = tthresh3;
            }
        }
    }
}
```