Лабораторная работа №4	M3137	2023
OpenMP	Грушевский Георгий Романович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: работа выполнена на C++, компилятор gcc. Все запускалось на AMD Ryzen 5 5500U (6 ядер, 12 потоков).

Описание

1. Изучить конструкции OpenMP для распараллеливания вычислений.
2. Написать программу, решающую поставленную задачу, любого из вариантов сложности.
3. Провести серию экспериментов по измерению времени работы программы.

Вариант

Я выполнил Hard-вариант лабораторной работы

Описание OpenMP

OpenMP - стандарт для многопоточных вычислений на C/C++. Управление распараллеливанием происходит с помощью директив компилятора (`#pragma omp`).

Директива, обозначающая что следующий блок кода должен быть исполнен параллельно - `parallel`. Далее можно задать число потоков

num_threads(<number>). Для указания глобальных переменных, видных потоку, нужно поставить default(none | shared). Shared означает, что все переменные, видимые из этого места в коде, совместно используются потоками. Указав none, нам нужно отдельно прописать shared(<var>) и private(<var>) переменные. Общий шаблон (из спецификации):

```
#pragma omp parallel for default(shared) firstprivate(i)\
private(x) private(r) lastprivate(i)
```

В данной работе будет достаточно только default(shared).

Параллельный блок в моей программе будет состоять из нескольких вложенных циклов for. Для распараллеливания цикла for существует директива:

```
#pragma omp for schedule(type[, chunk_size])
```

Данная директива распределяет итерации цикла for между потоками - каждому из них отводится некоторое количество (в зависимости от типа schedule и chunk_size) итераций.

Описание написанного кода

Согласно условию, программа должна преобразовывать изображение в четырехцветное - то есть, разбивать пиксели на 4 группы. Это предлагается делать методом Оцу.

Метод Оцу построен на максимизации межклассовой дисперсии итогового изображения по значениям порогов. Она вычисляется по следующей формуле:

$$\sigma^2 = \sum_{class=1}^{number\ of\ classes} P(class)\,(\mu(class) - \mu)^2$$

Для каждого класса нужно уметь вычислять вероятность принадлежности пикселя изображения к нему и матожидание яркости в кластере. Это можно делать за $O(1)$.

Выполним предподсчет двух массивов:

1. `vector<uint32_t> prob(256)` - фактически, массив префиксных сумм гистограммы, иначе - функция вероятности $P(a \leq x)$, помноженная на суммарное количество пикселей. Тогда вероятность принадлежности пикселя отрезку яркости (a, b] будет равна `prob[b] - prob[a].`
2. `vector<uint32_t> exp(256)` - массив, i-е значение которого есть матожидание яркости пикселя на отрезке [0, i]. В силу линейности матожидания, на произвольном отрезке яркости (a, b] матожидание считается как `exp[b] - exp[a].`

Код, выполняющий предподсчет:

```
std::vector<uint32_t> prob(256, 0);
std::vector<uint32_t> exp(256, 0);

prob[0] = histogram[0];
for (int t = 1; t < 256; ++t) {
   prob[t] = prob[t - 1] + histogram[t];
   exp[t] = t * histogram[t] + exp[t - 1];
}
```

Теперь переберем все значения порогов. Параллельные конструкции в коде опущены. В программе эти вычисления вынесены в функцию

`calculate_threshold`.Также заметим, что дисперсию можно считать с точностью до домножения на число пикселей, поэтому достаточно просто оперировать значениями массива `prob`.

```
double deviation = 0;
double avg = (double) exp.back() / n;

for (int th1 = 0; th1 < 253; th1++) {
    /* здесь переменные для обновления ответа */
    for (int th2 = th1 + 1; th2 < 254; th2++) {
        for (int th3 = th2 + 1; th3 < 255; th3++) {
            uint32_t p1, p2, p3, p4;
            double m1, m2, m3, m4;

            p1 = prob[th1];
            p2 = prob[th2] - prob[th1];
            p3 = prob[th3] - prob[th2];
            p4 = prob.back() - prob[th3];

            m1 = (double) exp[th1] / prob[th1];
            m2 = (double) (exp[th2] - exp[th1]) / (prob[th2] - prob[th1]);
            m3 = (double) (exp[th3] - exp[th2]) / (prob[th3] - prob[th2]);
            m4 = (double) (exp.back() - exp[th3]) / (prob.back() - prob[th3]);

            double cur_deviation = p1 * (m1 - avg) * (m1 - avg) +
                                   p2 * (m2 - avg) * (m2 - avg) +
                                   p3 * (m3 - avg) * (m3 - avg) +
                                   p4 * (m4 - avg) * (m4 - avg);
            /* обновление ответа */
            }
        }
   }
}
```

Заметим, что последовательные вычисления независимы, что даст ускорение при исполнении команд на superscalar.

Теперь поговорим про распараллеливание наших вычислений. Эффективнее всего будет использовать встроенное решение для распараллеливания `for` - одноименную директиву. Конструкции OpenMP в коде выглядят следующим образом:

```
#pragma omp parallel num_threads(num_thr) default(shared)
{
    std::vector<uint32_t> thread_private_ans(3, 0);
    double thread_private_dev = 0;

#pragma omp for schedule(dynamic)
    for (int th1 = 0; th1 < 253; th1++) {
        for (int th2 = th1 + 1; th2 < 254; th2++) {
            for (int th3 = th2 + 1; th3 < 255; th3++) {
                /* вычисления и обновление ответа внутри потока */
            }
        }
    }
#pragma omp critical
    {
       /* синхронизация ответа между всеми потоками */
    }
}
```

`default(shared)` потому, что этот код внутри функции, в аргументы которой передаются только параметры, совместно используемые всеми потоками. Также каждый из потоков имеет приватный массив ответов. После выполнения всех вычислений потоки поочередно сравнивают свои ответы и “выбирают” оптимальный в блоке директивы `critical.`

Результат работы программы

```
Time (1 thread(s)): 217.437 ms

Time (2 thread(s)): 89.625 ms

Time (3 thread(s)): 60.625 ms

Time (4 thread(s)): 48.9375 ms

Time (5 thread(s)): 41.1875 ms

Time (6 thread(s)): 37.4375 ms
```

```
Time (7 thread(s)): 35.9375 ms

Time (8 thread(s)): 32.5 ms

Time (9 thread(s)): 31.4375 ms

Time (10 thread(s)): 30.1875 ms

Time (11 thread(s)): 29.5625 ms

Time (12 thread(s)): 29.4375 ms

77

130

187
```

Результат работы на тестовом файле из условия

Экспериментальная часть

Для большего масштаба и точности я запускал программу в самом энерго-экономичном режиме Windows 10. Так, например, легче отследить оптимальное количество потоков. Все замеры времени усредняются по 16 значениям.

В принципе, нет смысла создавать больше потоков, чем есть логических ядер на машине (в моем случае 12). Поэтому, на всех графиках оптимум должен находится в окрестности 12 потоков.

Для замера времени я создал отдельный метод, возвращающий время работы:

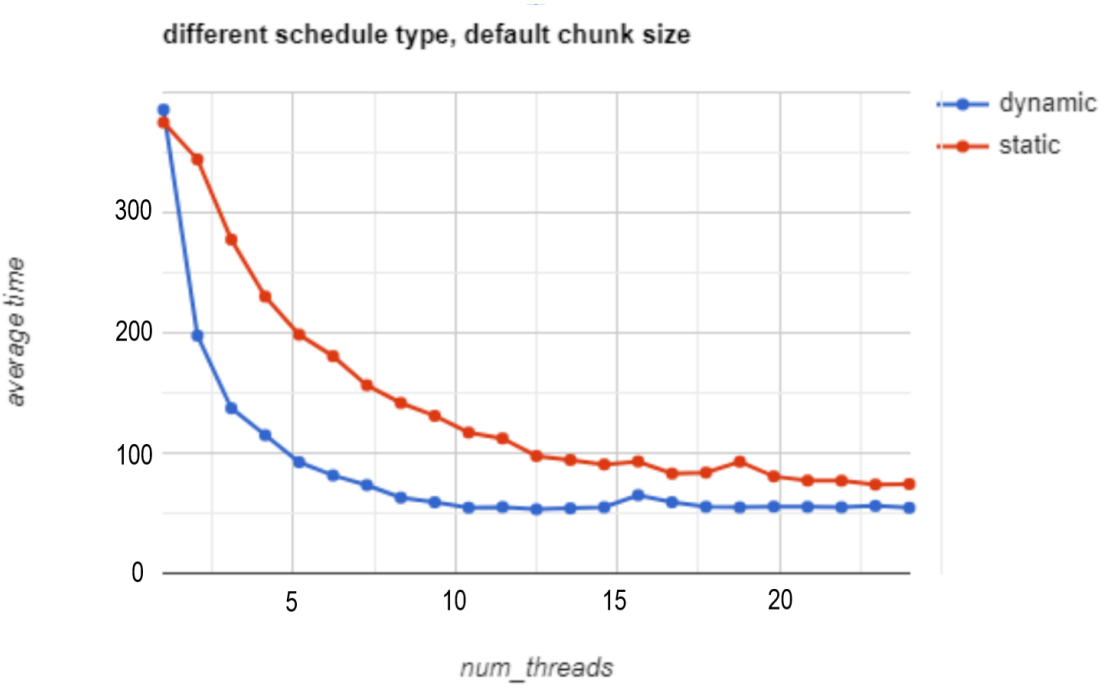
```
double measure_time(uint32_t n, std::vector<uint32_t> &exp,
                    std::vector<uint32_t> &prob,
                    std::vector<uint32_t> &answer, int num_thr) {
   double avg_time = 0;
   for (int i = 0; i < 16; ++i) {
       double start = omp_get_wtime();

       calculate_threshold(n, exp, prob, answer, num_thr);

       avg_time += (omp_get_wtime() - start);
   }
   return avg_time * 1000 / 16;
}
```

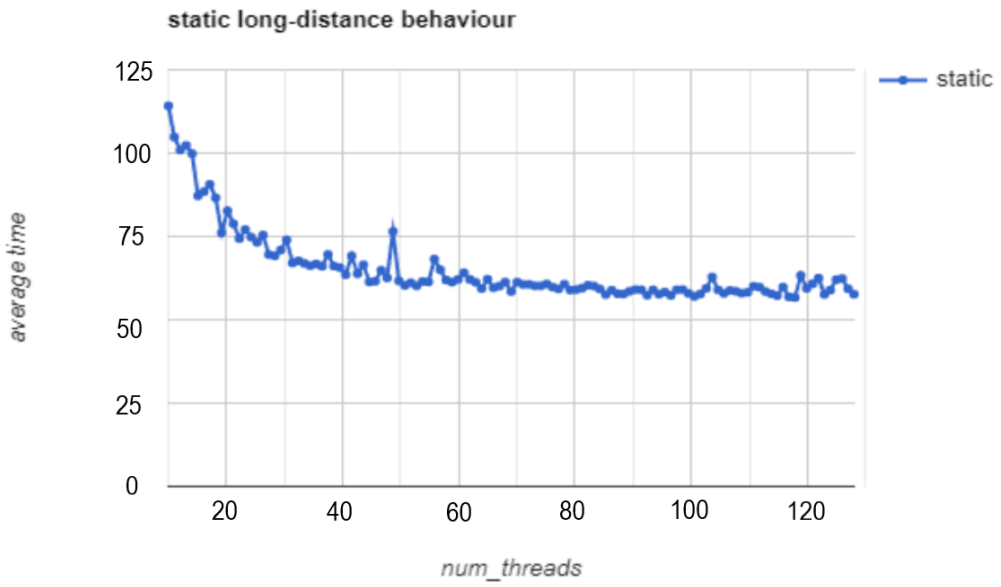
Время умножается на 1000 для перевода в миллисекунды.

1. График времени исполнения при фиксированном типе schedule и изменяемом числе потоков:



Очевидно преимущество dynamic - время исполнения в 2 раза меньше, чем у static (в некоторых точках), также оно довольно быстро стабилизируется с возрастанием числа потоков (как и ожидалось, в окрестности 12).

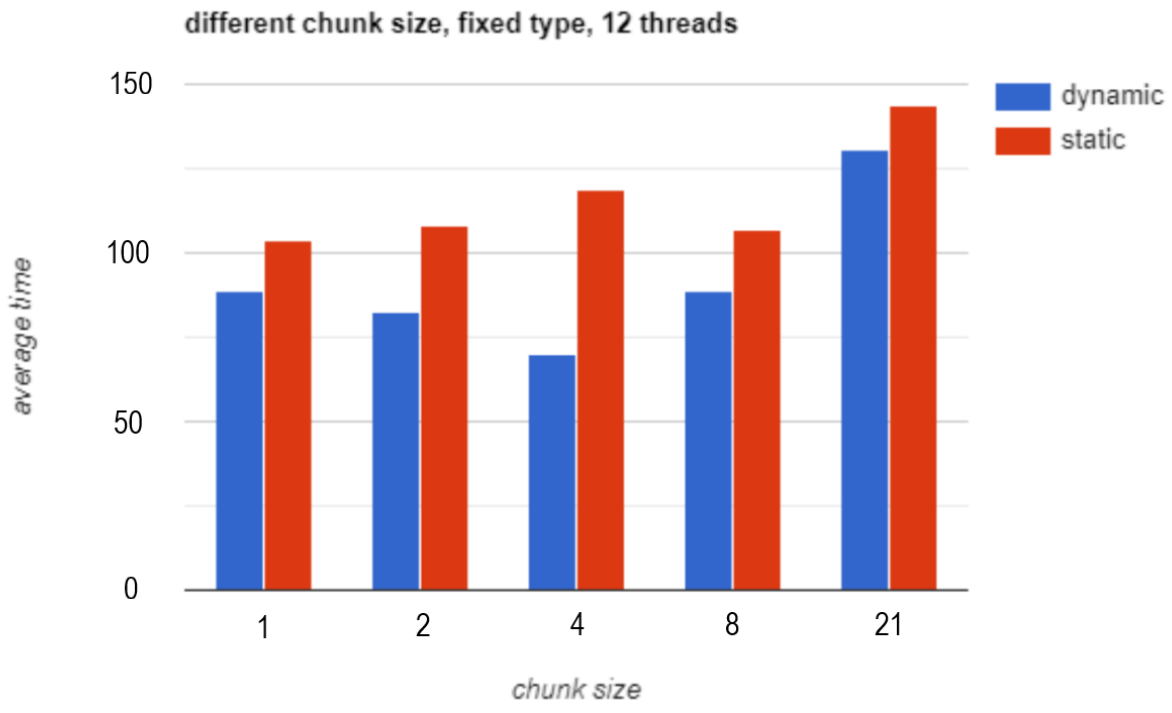
Проверим, как ведет себя static на “патологическом” увеличении числа потоков:



Первое что оказалось характерно для static - время выполнения явно убывает до num_threads = 40. Это является следствием того, что итерации нашего цикла имеют разный размер, и в итоге один из потоков получает большой кусок вычислений (больше, чем другие). С увеличением числа потоков этот большой кусок уменьшается в размерах, почему время и убывает так долго. Далее колебания времени становятся минимальны, так как потоки получают по очень малому количеству итераций, и начинают “плотно” исполняться друг за другом. Также, это свидетельствует о том, что выполнение итерации цикла занимает на порядок большее время, чем создание потока.

К слову, я запускал static на количестве потоков, большем количества итераций цикла, но никаких изменений это не дало.

2. График времени исполнения при фиксированном типе schedule и разных chunk size

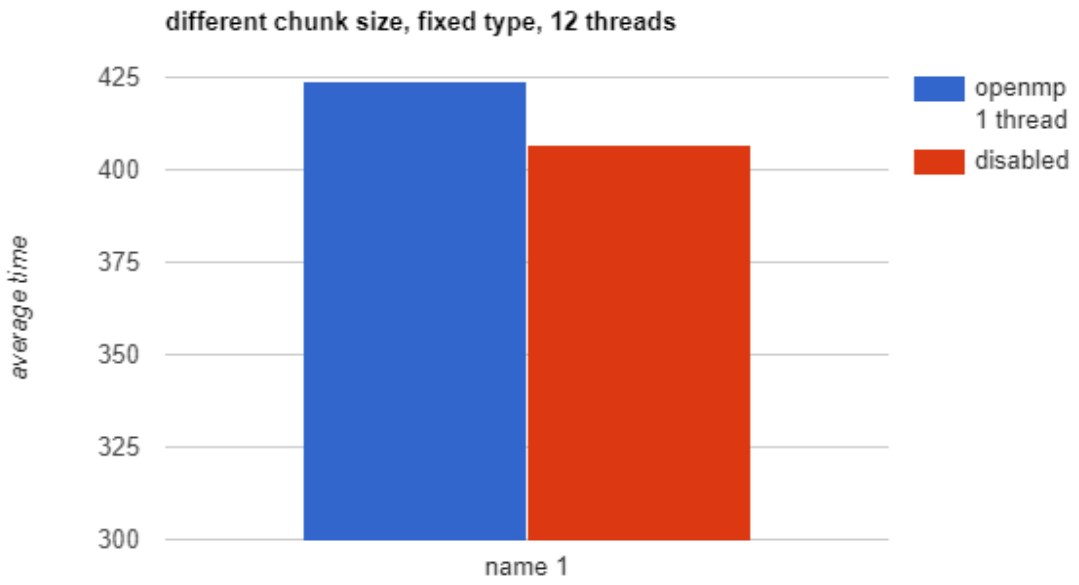


Количество потоков выбрано оптимальным - 12. Значения chunk size не больше 21 = 254 // 12.

Как можно наблюдать по гистограмме, у dynamic полное преимущество по скорости над static. Это получилось потому, что количество действий в итерации распараллеленного цикла убывает с ростом номера итерации. По этой же причине на chunk_size, равном 1, 2, 4, 8 не получились одинаковые ответы - все итерации разного размера, и поэтому то, является ли размер чанка точным (ну почти) делителем общего количества, ничего не зависит.

Думаю, получилось бы распараллелить эти вычисления эффективнее, если бы такой цикл пробегал по всем сочетаниям из 256 по 3 - тогда его итерации были бы одинаковыми по объему. Предполагаю, что в таком случае static давал бы небольшой выигрыш над dynamic (не нужно было бы тратить время на распределение между свободными потоками). Однако, этот способ был бы рекурсивным (если, конечно, без предподсчета сочетаний) и исполнялся бы, наверное, даже дольше.

Сравнение выключенного OpenMP и OpenMP с одним потоком



Логично было предположить, что подключение OpenMP занимает время. Поэтому без него работает быстрее, чем с ним на 1 потоке.

Источники

1. https://www.rapidtables.com/tools/line-graph.html
2. https://learn.microsoft.com/ru-ru/cpp/parallel/openmp/1-introduction?view=msvc-170
3. https://www.openmp.org//wp-content/uploads/cspec20.pdf
4. https://youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG

Листинг кода

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

char convert(std::vector<uint32_t> &a, uint8_t pixel) {
   if (pixel <= a[0]) {
       return 0;
   } else if (pixel <= a[1]) {
       return 84;
   } else if (pixel <= a[2]) {
       return -86;
   } else {
       return -1;
   }
}

void calculate_threshold(uint32_t n, std::vector<uint32_t> &exp,
std::vector<uint32_t> &prob,
                         std::vector<uint32_t> &answer, int num_thr) {

   double deviation = 0;
   double avg = (double) exp.back() / n;

#pragma omp parallel num_threads(num_thr) default(shared)
```

```
    {
        std::vector<uint32_t> thread_private_ans(3, 0);
        double thread_private_dev = 0;
#pragma omp for schedule(dynamic)
        for (int th1 = 0; th1 < 254; th1++) {
            for (int th2 = th1 + 1; th2 < 255; th2++) {
                for (int th3 = th2 + 1; th3 < 256; th3++) {
                    uint32_t p1, p2, p3, p4;
                    double m1, m2, m3, m4;

                    p1 = prob[th1];
                    p2 = prob[th2] - prob[th1];
                    p3 = prob[th3] - prob[th2];
                    p4 = prob.back() - prob[th3];

                    m1 = (double) exp[th1] / prob[th1];
                    m2 = (double) (exp[th2] - exp[th1]) / (prob[th2] -
prob[th1]);
                    m3 = (double) (exp[th3] - exp[th2]) / (prob[th3] -
prob[th2]);
                    m4 = (double) (exp.back() - exp[th3]) / (prob.back() -
prob[th3]);

                    double cur_deviation = p1 * (m1 - avg) * (m1 - avg) +
                                           p2 * (m2 - avg) * (m2 - avg) +
                                           p3 * (m3 - avg) * (m3 - avg) +
                                           p4 * (m4 - avg) * (m4 - avg);

                    if (cur_deviation > thread_private_dev) {
                        thread_private_dev = cur_deviation;
                        thread_private_ans[0] = th1;
                        thread_private_ans[1] = th2;
                        thread_private_ans[2] = th3;
                    }
                }
            }
        }
#pragma omp critical
        {
            if (thread_private_dev > deviation) {
                deviation = thread_private_dev;
                answer[0] = thread_private_ans[0];
                answer[1] = thread_private_ans[1];
                answer[2] = thread_private_ans[2];
            }
        }
    }
}

void calculate_threshold_no_omp(uint32_t n, std::vector<uint32_t> &exp,
```

```
std::vector<uint32_t> &prob,
                                std::vector<uint32_t> &answer) {

   double deviation = 0;
   double avg = (double) exp.back() / n;

   for (int th1 = 0; th1 < 254; th1++) {
       for (int th2 = th1 + 1; th2 < 255; th2++) {
           for (int th3 = th2 + 1; th3 < 256; th3++) {
               uint32_t p1, p2, p3, p4;
               double m1, m2, m3, m4;

               p1 = prob[th1];
               p2 = prob[th2] - prob[th1];
               p3 = prob[th3] - prob[th2];
               p4 = prob.back() - prob[th3];

               m1 = (double) exp[th1] / prob[th1];
               m2 = (double) (exp[th2] - exp[th1]) / (prob[th2] - prob[th1]);
               m3 = (double) (exp[th3] - exp[th2]) / (prob[th3] - prob[th2]);
               m4 = (double) (exp.back() - exp[th3]) / (prob.back() -
prob[th3]);

               double cur_deviation = p1 * (m1 - avg) * (m1 - avg) +
                                      p2 * (m2 - avg) * (m2 - avg) +
                                      p3 * (m3 - avg) * (m3 - avg) +
                                      p4 * (m4 - avg) * (m4 - avg);

               if (cur_deviation > deviation) {
                   deviation = cur_deviation;
                   answer[0] = th1;
                   answer[1] = th2;
                   answer[2] = th3;
               }
           }
       }
   }

}

double measure_time(uint32_t n, std::vector<uint32_t> &exp,
std::vector<uint32_t> &prob,
                    std::vector<uint32_t> &answer, int num_thr) {
   double avg_time = 0;
   for (int i = 0; i < 16; ++i) {
       double start = omp_get_wtime();

       num_thr == -1 ? calculate_threshold_no_omp(n, exp, prob, answer)
                     : calculate_threshold(n, exp, prob, answer, num_thr);
```

```
        avg_time += (omp_get_wtime() - start);
    }
    return avg_time * 1000 / 16;
}

int main(int argc, char *argv[]) {
    std::ifstream in;
    in.open(argv[2]);

    if (!in.is_open()) {
        std::cout << "error while opening input file" << std::endl;
        exit(1);
    }

    std::ofstream out;
    out.open(argv[3]);

    if (!out.is_open()) {
        std::cout << "error while opening output file" << std::endl;
        in.close();
        exit(1);
    }

    std::string s;
    getline(in, s);

    uint32_t height, width;
    in >> width >> height;
    uint32_t n = height * width;

    getline(in, s);

    std::vector<std::vector<uint8_t>> image(width, std::vector<uint8_t>(height));
    std::vector<uint32_t> histogram(256, 0);
    std::vector<uint32_t> prob(256, 0);
    std::vector<uint32_t> exp(256, 0);

    for (int x = 0; x < height; ++x) {
        for (int y = 0; y < width; ++y) {
            uint8_t pixel;
            in.read((char *) &pixel, sizeof(uint8_t));
            image[y][x] = pixel;
            histogram[pixel]++;
        }
    }
    in.close();

    prob[0] = histogram[0];
    for (int t = 1; t < 256; ++t) {
        prob[t] = prob[t - 1] + histogram[t];
```

```
        exp[t] = t * histogram[t] + exp[t - 1];
    }

    std::vector<uint32_t> answer(3, UINT32_MAX);

    int num_thr = std::stoi(argv[1]);
    if (num_thr < -1) {
        std::cout << "invalid threads number" << std::endl;
        out.close();
        exit(1);
    }

    printf("Time (%i thread(s)): %g ms\n", num_thr,
           measure_time(n, exp, prob, answer, num_thr));


    std::cout << answer[0] << std::endl;
    std::cout << answer[1] << std::endl;
    std::cout << answer[2] << std::endl;

    out << "P5\n";
    out << width << " " << height << "\n";
    out << "255\n";

    for (int x = 0; x < height; x++) {
        for (int y = 0; y < width; y++) {
            char z = convert(answer, image[y][x]);
            out.write(&z, sizeof(char));
        }
    }
    out.close();
}
```