|  |  |  |
| --- | --- | --- |
| Лабораторная работа №4 | M3136 | 2023 |
| OpenMP | Корнилович Михаил Антонович | |
|  |

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

**Инструментарий:** C++ / MinGW.org GCC-6.3.0-1. Стандарт OpenMP 2.0.

**Описание конструкций OpenMP для распараллеливания команд.**

Перед всеми блоками кода, которые надо распараллелить, пишется #pragma omp parallel. Мы можем добавить к этому блоку if statement, который позволит нам запускать блок кода параллельно при условии, что в if statement значение не 0. Также можно дописать for, которое означает, что мы хотим распараллелить цикл for. Есть возможность указать количество thread-ов, которое мы хотим использовать при исполнении данного блока кода. Это делается с помощью num\_threads(threads\_count) (в скобках указывается количество потоков). Также есть параметр schedule, который распределяет работу между thread-ами так, чтобы минимизировать время ожидания потоков у барьера. В нём мы можем указать, как будет распределяться задачи динамично (dynamic) или статично (static), а также можно установить размер одного подмножество задач, который будет выполнять thread. Есть параметр default, который позволяет повлиять на data-sharing. Есть два варианта shared, который превращает все переменные в области видимости в общие, а есть none, который даёт возможность удалить все внешние (относительно блока кода, который мы параллелим) переменные. А затем также можно указать вручную, какие переменные будут общими (с помощью параметра shared), а какие приватными (с помощью private).

**Описание работы написанного кода.**

Сначала необходимо считать исходное изображение. Для жто напишем функцию, которая будем считывать картинку формата pgm (pnm P5) и возвращать массив байтов, где каждый байт хранит информацию о цвете одного пикселя. Также будем выкидывать ошибку в случае, если картинка не соответствует формату.

struct image\_size {  
 int width;  
 int height;  
};  
  
std::pair<image\_size, std::vector<uint8\_t>> read\_from\_file(const std::string &fileName) {  
 std::ifstream file(fileName);  
 std::vector<uint8\_t> data;  
 int width;  
 int height;  
 std::string str;  
 file >> str;  
 if (str != "P5") {  
 throw std::runtime\_error("Invalid file format, expected P5");  
 }  
 file >> width >> height;  
 file >> str;  
 if (str != "255") {  
 throw std::runtime\_error("Invalid file format, expected 255");  
 }  
 data.resize(width \* height);  
 std::unique\_ptr<char[]> buffer(new char[height \* width]);  
 file.read(buffer.get(), width \* height);  
 for (int i = 0; i < height; ++i) {  
 for (int j = 0; j < width; ++j) {  
 data[i \* width + j] = (uint8\_t) buffer[i \* width + j];  
 }  
 }  
 file.close();  
 return {image\_size{width, height}, data};  
}

Листинг 1 – чтение из файла

После чего на основании исходного изображения сгенерируем его гистограмму. Для этого напишем функцию, которая принимает на вход исходное изображение и его размеры. После чего подсчитывается количество вхождений каждого байта. На выходе мы получаем гистограмму.

std::vector<int> generate\_histogram(std::vector<uint8\_t> &data, int width, int height) {  
 std::vector<int> histogram(256);  
 for (int i = 0; i < width \* height; ++i) {  
 ++histogram[data[i]];  
 }  
 return histogram;  
}

Листинг 2 – генерация гистограммы изображения.

После этого нам нужно посчитать пороги, но перед этим пред подсчитаем вероятности кластеров и их среднее значение (префиксные суммы). После чего нужно параллельно найти лучшую комбинацию порогов. Для этого мы будем использовать три вложенных цикла. Чтобы сделать это параллельно нам нужно указать количество потоков, которое подаётся в аргументах программы, параметр schedule с dynamic и с chunk\_size. Также установим параметр default на none и укажем нужные нам переменные через shared. И находим нужные пороги по алгоритму Оцу. При этом там, где мы записываем актуальные порги мы сделаем блок critical. При этом перед блоком critical мы напишем if, который проверяет больше ли текущая межкластерная дисперсия максимальной дисперсии, которая уже была найдена. И внутри critical мы таже напишем этот if. Внешний if нужен, чтобы программа реже попадала в блок critical, тем самым потоки не будут лишний раз ждать других потоков. Внутренний if нужен для синхронизации потоков.

struct Thresholds {  
 uint8\_t f0;  
 uint8\_t f1;  
 uint8\_t f2;  
};  
  
Thresholds calculate\_thresholds(std::vector<int> &histogram, int width, int height, int threadCount, int chunkSize) {  
 std::vector<double> q = precalculate\_chance\_for\_thresholds(histogram, width, height);  
 std::vector<double> u = precalculate\_average(histogram, width, height);  
 double v\_max = 0;  
 Thresholds answer;  
 omp\_set\_dynamic(0);  
#pragma omp parallel for if (threadCount != -1) num\_threads(threadCount) schedule(dynamic, chunkSize) default(none) shared(answer, v\_max, q, u, chunkSize)  
 for (int f0 = 0; f0 < 254; ++f0) {  
 for (int f1 = f0 + 1; f1 < 255; ++f1) {  
 for (int f2 = f1 + 1; f2 < 256; ++f2) {  
 double q0 = q[f0] - q[0];  
 double q1 = q[f1] - q[f0];  
 double q2 = q[f2] - q[f1];  
 double q3 = q[255] - q[f2];  
 double u0 = (u[f0] - u[0]) / q0;  
 double u1 = (u[f1] - u[f0]) / q1;  
 double u2 = (u[f2] - u[f1]) / q2;  
 double u3 = (u[255] - u[f2]) / q3;  
 double u = q0 \* u0 + q1 \* u1 + q2 \* u2 + q3 \* u3;  
 double v = q0 \* (u0 - u) \* (u0 - u) + q1 \* (u1 - u) \* (u1 - u)  
 + q2 \* (u2 - u) \* (u2 - u) + q3 \* (u3 - u) \* (u3 - u);  
 if (v > v\_max) {  
#pragma omp critical  
 if (v > v\_max) {  
 answer = Thresholds{(uint8\_t)f0,(uint8\_t) f1, (uint8\_t)f2};  
 v\_max = v;  
 }  
 }  
 }  
 }  
 }  
 return answer;  
}

Листинг 3 – нахождение порогов

После этого нужно получить новое изображение. Сделаем это также многопоточно. Полученный массив мы будем записывать в файл.

std::vector<char> get\_new\_image(std::vector<uint8\_t> &data, Thresholds t, int threadCount, int chunkSize) {  
 std::vector<char> new\_image(data.size());  
 omp\_set\_dynamic(0);  
#pragma omp parallel for if (threadCount != -1) num\_threads(threadCount) shared(new\_image)  
 for (int i = 0; i < data.size(); ++i) {  
 char ch;  
 if (data[i] <= t.f0) {  
 ch = (char)0;  
 } else if (data[i] <= t.f1) {  
 ch = (char)84;  
 } else if (data[i] <= t.f2) {  
 ch = (char)170;  
 } else {  
 ch = (char)255;  
 }  
 new\_image[i] = ch;  
 }  
 return new\_image;  
}  
  
  
void generate(std::vector<uint8\_t> &data, int width, int height, int threadCount, const std::string &outputFileName, int chunkSize) {  
 std::vector<int> histogram = generate\_histogram(data, width, height);  
 Thresholds t = calculate\_thresholds(histogram, width, height, threadCount, chunkSize);  
 std::cout << get\_format\_string("%u %u %u\n", t.f0, t.f1, t.f2);  
 std::vector<char> new\_image = get\_new\_image(data, t, threadCount, chunkSize);  
 std::ofstream file(outputFileName);  
 file << "P5" << std::endl;  
 file << width << " " << height << std::endl;  
 file << 255 << std::endl;  
 file.write(new\_image.data(), new\_image.size());  
 file.close();  
}

Листинг 4 – генерация нового изображения и запись в файл

**Результат работы написанной программы**

Процессор – Intel Core i9 – 12900HK

Лог программы:

77 130 187

Time (20 thread(s)): 3.00002 ms

**Экспериментальная часть**

Чтобы сделать замеры, я написал скрипт на python, который запускает программу и записывает всё excel таблицу в удобном формате (executeTest.py). Каждое значение получено как среднее арифметическое за 5 тестов.

Рисунок 1 – График времени с разным количеством потоков (все schedule static)

Рисунок 2 - График времени с разным количеством потоков (все schedule dynamic)

Рисунок 3 - График времени с разным количеством потоков (schedule histogram – static, calculate thresholds – dynamic, generate new image - static)

Рисунок 4 - График времени с разным количеством потоков (schedule histogram – guided, calculate thresholds – dynamic, generate new image - static)