САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

**«OpenMP»**

Выполнил: Тяпкин Пётр Алексеевич

Номер ИСУ: 334844

студ. гр. M3138

Санкт-Петербург

2021

**Цель работы:** знакомство со стандартом OpenMP.

**Инструментарий и требования к работе:**рекомендуется использовать C, C++. Возможно использовать Python и Java. Стандарт OpenMP 2.0.

**Теоретическая часть**

OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках C, C++ и Fortran. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Библиотека OpenMP часто используется в математических вычислениях, т.к. позволяет очень быстро и без особого труда распараллелить вашу программу.

Существует множество разновидностей параллельных вычислительных систем — многоядерные/многопроцессорные компьютеры, кластеры, системы на видеокартах, программируемые интегральные схемы и т.д. Библиотека OpenMP подходит только для программирования систем с общей памятью, при этом используется параллелизм потоков. Потоки создаются в рамках единственного процесса и имеют свою собственную память. Кроме того, все потоки имеют доступ к памяти процесса.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой ведущий (master) поток создаёт набор ведомых потоков, и задача распределяется между ними.

Задачи, выполняемые потоками параллельно, так же, как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка.

Директивы можно разделить на 3 категории: определение параллельной секции, разделение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов - клауз. Отдельно специфицируются клаузы для назначения классов переменных, которые могут быть атрибутами различных директив.

Для С/С++ директивы имеют следующий вид:

#pragma omp конструкция [условие [условие]...]

Параллельные регионы (секции) являются основным понятием в OpenMP. Именно там, где задан этот регион программа исполняется параллельно. Как только компилятор встречает #pragma omp parallel, он вставляет инструкции для создания параллельных потоков. Количество порождаемых потоков для параллельных областей контролируется через переменную окружения OMP\_NUM\_THREADS, а также может задаваться через вызов функции внутри программы. Каждый порожденный поток исполняет блок код в структурном блоке. По умолчанию синхронизация между потоками отсутствует и поэтому последовательность выполнения конкретного оператора различными потоками не определена. После выполнения параллельного участка кода все потоки, кроме основного завершаются, и только основной поток продолжает исполняться, но уже один.

Каждый поток имеет свой уникальный номер, который изменяется от 0 (для основного потока) до количества потоков – 1. С помощью функций OMP\_GET\_THREAD\_NUM и OMP\_GET\_NUM\_THREADS поток может узнать свой номер и общее число нитей, а затем выполнять свою часть работы в зависимости от своего номера.

Самый популярный способ распределения задач в OpenMP — параллельный цикл, при этом между итерациями цикла не должно быть зависимостей. Такой цикл называется векторизуемым (его итерации можно поделить между потоками и выполнить независимо друг от друга). Параллельный цикл позволяет задать условие schedule.

Данное условие контролирует то, как работа будет распределяться между потоками.

schedule(тип [, размер блока])

Размер блока задает размер каждого пакета на обработку потоком (количество итераций). Тип расписания может принимать следующие значение:

* static – итерации равномерно распределяются по потокам. Если при этом задан еще и размер блока, то все итерации блоками заданного размера циклически распределяются между потоками.

Статическое распределение работы эффективно, когда время выполнения итераций равно, или приблизительно равно.

* dynamic – работа распределяется пакетами заданного размера (по умолчанию размер равен 1) между потоками. Как только какой-либо из потоков заканчивает обработку своей порции данных, он захватывает следующую.

При этом подходе несколько большие накладные расходы, но можно добиться лучшей балансировки загрузки между потоками.

* guided – данный тип распределения работы аналогичен предыдущему, за тем исключением, что размер блока изменяется динамически в зависимости от того, сколько необработанных итераций осталось. Размер блока постепенно уменьшается вплоть до указанного значения.

При таком подходе можно достичь хорошей балансировки при меньших накладных расходах.

* runtime – тип распределения определяется в момент выполнения программы.

## Классы переменных

В OpenMP переменные в параллельных областях программы разделяются на два основных класса shared и private.

Условие shared указывает на то, что все перечисленные переменные будут разделяться между потоками. Все потоки будут иметь доступ к одной и той же области памяти.

Условие private указывает на то, что каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.

Отдельные правила определяют поведение переменных при входе и выходе из параллельной области или параллельного цикла: reduction, firstprivate, lastprivate, copyin.

Все переменные, созданные до директивы parallel являются общими (shared) для всех потоков. Переменные, созданные внутри потока, являются локальными (private) и доступны только текущему потоку. При изменении общей переменной одновременно несколькими потоками возникает состояние гонок (мы не можем гарантировать какой-либо конкретный порядок записи и, следовательно, результат).

**Алгоритм**

Я решал hard версию - автоматическая контрастность изображения. Задача состоит в том, что что дано изображение, которое может иметь плохую контрастность: используется не весь диапазон значений, а только его часть. И нужно изменить значения пикселей таким образом, чтобы получить максимальную контрастность: растянуть диапазон значений до [0; 255], но при этом не изменить оттенки (то есть в цветных изображениях нужно одинаково изменять каналы R, G и B).

Так же нам дан коэффициент. И при вычислении растяжения нужно игнорировать некоторую долю (по количеству) самых тёмных и самых светлых точек (для RGB в каждом канале отдельно). Это позволяет игнорировать шум, который незаметен глазу, но мешает автоматической настройке контрастности. Растяжение диапазона следует выполнять с насыщением, чтобы проигнорированные пиксели не вышли за границы [0; 255].

1. Считаем изображение, то есть width, height, type и набор пикселей. Кол-во пикселей будет равно width \* height и умножить на 3, если изображение RGB, то есть type == “P6”.
2. Отсортируем пиксели по каждому каналу отдельно. Я использовал для этого сортировку подсчетом, которую распараллелил с помощью параллельных секций и параллельного цикла.
3. Проигнорируем в зависимости от коэффициента, какое-то кол-во самых темных и светлых, и найдем среди оставшихся минимум (mn) и максимум (mx).
4. Теперь у нас есть диапазон [mn, mx], а мы хотим получить [0, 255]. Тогда значение пикселя будет изменятся по следующей формуле



Формула следует из геометрических соображений подобия.

Для лучшего понимания см. рисунок 1.

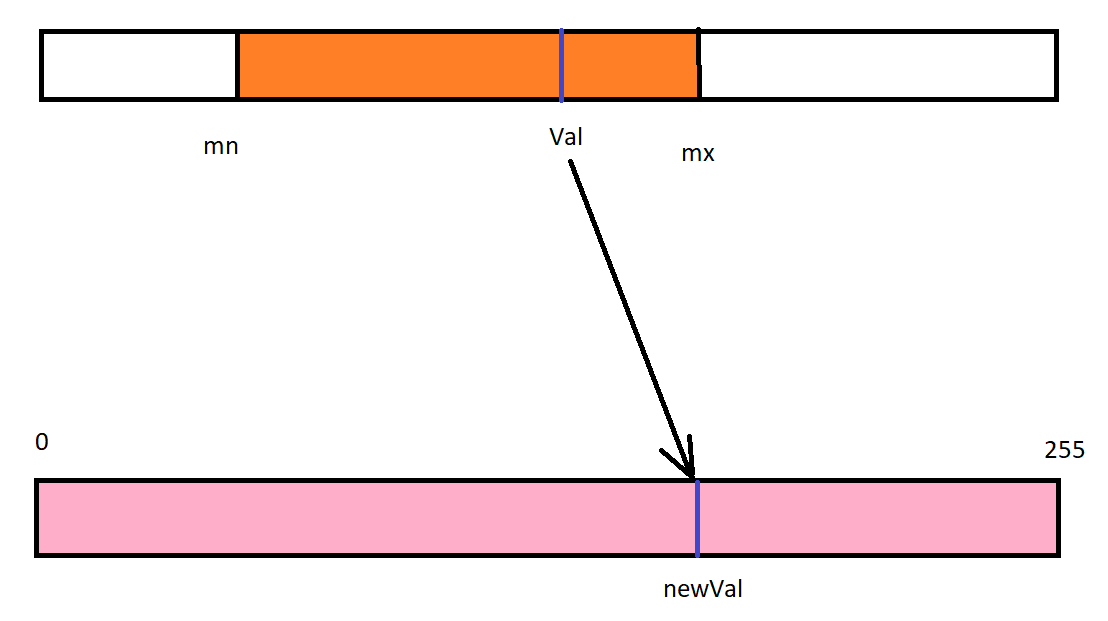


Рисунок № 1 – Растяжение диапазона

1. Учтем, что растяжение диапазона следует выполнять с насыщением, чтобы проигнорированные пиксели не вышли за границы [0; 255]. И поменяем значения пикселей в исходном наборе, согласно формуле, приведенной выше. Этот изменение пикселей я сделал с использованием параллельного цикла, так как между итерациями цикла нет никаких зависимостей.

**Практическая часть**

Я реализовал алгоритм, приведенный в теории на языке С++.

1. Я считал данные проверил, что из нужного кол-ва и что они корректны в рамках данной задачи.
2. Проверил, что изображение состоит из одного цвета. Так как в таком случае оно не обрабатывается (не меняется) – функция check().
3. По умолчанию кол-во потоков 4. И если это нужно, то меняется.
4. Первая часть алгоритма сортировка подсчетом. Я разбил массив пикселей image на части, кол-во частей равно кол-ву потоков. И каждый поток обработал свою часть массива, для этого я создал параллельную. секцию. И сложил данные в вектор cur. Это нужно для того чтобы не случилось состояние гонки (англ. race condition).
5. Затем сложил все данные из cur в наш массив sort, в котором уже будут лежать данные в отсортированном виде. Это сделал с помощью параллельного цикла. (cnt[i][j] – кол-во пикселей значение которых i в j канале).
6. Затем подсчитал, какое кол-во самых светлых и темных пикселей нужно проигнорировать по каждому каналу. И среди оставшихся нашел минимальное и максимальное значение. Теперь [mn, mx] это наш диапазон, который нам нужно расширить.
7. С помощью параллельного цикла изменяем значение пикселей в массиве image, по формуле из теории. И не забываем о насыщении.

В алгоритме присутствует #ifdef \_OPENM, это нужно чтобы, когда я компилировал без –fopenmp, то алгоритм тоже корректно работал.

Сравним результаты работы программы, с разными параметрами. Все тесты проводились на картинке, которая будет лежать в архиве вместе с исходным кодом.

1. с разными параметрами schedule и разным числом потоков (см график № 1, 2, 3).

График № 1 - Static 100 vs Dynamic 100

График № 2 - Static 1 vs Dynamic 1

График № 3 - Static 512 vs Dynamic 512

1. С выключенным openmp и с включенным с 1 потоком (см. график № 4).

График № 4 - OpenMP vs without OpenMP

**Листинг**

Компилятор Microsoft Visual C++ 2017. Нужно указывать опцию сборки

-fopenmp или включить флаг в Visual Studio.

hw5.cpp

#include <chrono>

#include <iostream>

#include <fstream>

#include <omp.h>

#include <string>

#include <vector>

#include <cassert>

using namespace std;

int width = -1, height = -1, max\_val = -1;

string type = "";

vector<unsigned char> image;

int cntСhannels() {

int countСhannels = 1;

if (type == "P6") {

countСhannels = 3;

}

return countСhannels;

}

int getSzImage() {

return width \* height \* cntСhannels();

}

void read(string input\_file) {

ifstream reader(input\_file, ios::binary);

if (!reader.is\_open()) {

cerr << "Cannot open input\_file";

exit(1);

}

getline(reader, type);

if (type != "P5" && type != "P6") {

cerr << "Incorrect type " << type << endl;

exit(1);

}

reader >> width >> height >> max\_val;

string skip;

getline(reader, skip);

image = vector<unsigned char>(getSzImage());

for (int i = 0; i < getSzImage(); i++) {

reader.read((char\*)&image[i], sizeof(unsigned char));

}

reader.close();

}

void write(string output\_file) {

ofstream writer(output\_file, ios::binary);

if (!writer.is\_open()) {

cerr << "Cannot open output\_file";

exit(1);

}

writer << type << endl;

writer << width << " " << height << endl;

writer << 255 << endl;

for (int i = 0; i < getSzImage(); i++) {

writer << (char)(image[i]);

}

writer.close();

}

bool check() {

bool ans = true;

for (int i = 0; i < getSzImage(); i++) {

if (i - cntСhannels() >= 0 && image[i] != image[i - cntСhannels()]) {

ans = false;

break;

}

}

return ans;

}

int sort[256][3];

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

// считывание входных данных

if (argc < 5) {

cerr << "Incorrect input";

return 0;

}

string argv1 = argv[1];

int threads = atoi(argv1.c\_str());

string input\_name = argv[2];

string output\_name = argv[3];

string argv4 = argv[4];

double coefficient = atof(argv4.c\_str());

if (!(0 <= coefficient && coefficient <= 0.5)) {

cerr << "Incorrect coefficient";

return 0;

}

int cntThreads = 4; // значение по умолчанию

#ifdef \_OPENMP

if (threads != 0) {

cntThreads = threads;

}

if (!(1 <= cntThreads && cntThreads <= 2048)) {

cerr << "Incorrect cntThreads";

return 0;

}

omp\_set\_num\_threads(cntThreads);

#endif

read(input\_name);

// проверка что изображение состоит из одного цвета

if (check()) {

float tim = 0.0;

#ifdef \_OPENMP

printf("Time (%i thread(s)): %g ms\n", cntThreads, tim);

#endif

write(output\_name);

return 0;

}

chrono::time\_point<chrono::high\_resolution\_clock> start = std::chrono::high\_resolution\_clock::now();

// сортировка подсчетом

#ifdef \_OPENMP

int sz = (getSzImage() + cntThreads - 1) / cntThreads;

vector<vector<vector<int>>> cur(cntThreads, vector<vector<int>>(256, vector<int>(3, 0)));

#pragma omp parallel

{

int idThread = omp\_get\_thread\_num();

for (int i = idThread \* sz; i < min((idThread + 1) \* sz, getSzImage()); i++) {

cur[idThread][image[i]][i % cntСhannels()]++;

}

}

for (int i = 0; i < cntThreads; i++) {

#pragma omp parallel for schedule(static)

for (int j = 0; j < 256; j++) {

sort[j][0] += cur[i][j][0];

sort[j][1] += cur[i][j][1];

sort[j][2] += cur[i][j][2];

}

}

#else

for (int i = 0; i < getSzImage(); i++) {

sort[image[i]][i % cntСhannels()]++;

}

#endif

// проверка data race

int sum = 0;

for (int i = 0; i < 256; i++) {

sum += sort[i][0] + sort[i][1] + sort[i][2];

}

if (sum != getSzImage()) {

cout << "Data race error";

exit(1);

}

// подсчет самых светлых и темных пикселей, которые нужно проигнорировать

// нахождение min и max значений пикселей, не считая игнорированные

int mn = 255, mx = 0;

for (int i = 0; i < cntСhannels(); i++) {

int ignoreL = 0;

int cntL = 0;

while (cntL < getSzImage() / cntСhannels() \* coefficient) {

cntL += sort[ignoreL++][i];

}

int ignoreR = 255;

int cntR = 0;

while (cntR < getSzImage() / cntСhannels() \* coefficient) {

cntR += sort[ignoreR--][i];

}

while (sort[ignoreL][i] == 0) {

ignoreL++;

}

while (sort[ignoreR][i] == 0) {

ignoreR--;

}

mn = min(mn, ignoreL);

mx = max(mx, ignoreR);

}

// изменение исходной картинки

#pragma omp parallel for schedule(static, 100)

for (int i = 0; i < getSzImage(); i++) {

int x = (image[i] - mn) \* (255.0 / (mx - mn));

x = max(x, 0);

x = min(x, 255);

image[i] = x;

}

// вывод

chrono::time\_point<chrono::high\_resolution\_clock> finish = std::chrono::high\_resolution\_clock::now();

float tim = ((float)chrono::duration\_cast<chrono::microseconds>(finish - start).count()) / 1000;

#ifdef \_OPENMP

printf("Time (%i thread(s)): %g ms\n", cntThreads, tim);

#endif

write(output\_name);

return 0;

}