САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил: Волков Егор Яковлевич

студ. гр. М3139

Санкт-Петербург

2020

Цель работы: знакомство со стандартом распараллеливания команд OpenMP.

Инструментарий и требования к работе: C++ с библиотекой <omp.h>

Теоретическая часть

Описание OpenMP

OpenMP позволяет добавлять в программы параллельность, не задумываясь о самостоятельном создании, синхронизации, удалении потоков, а просто указывая необходимые директивы компилятора. Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для распараллеливания участков кода.

Параллельная область объявляется с помощью директивы omp parallel с набором необязательных параметров, рассмотренных ниже.

После запуска программы создается единственный процесс с единственным master-потоком. Встретив параллельную область, master порождает несколько slave-потоков. Можно регулировать общее количество потоков с помощью функции omp_set_num_threads или переменной окружения OMP_NUM_THREADS. Число потоков по умолчанию равно количеству вычислительных ядер. После выполнения параллельного участка кода все потоки, кроме основного, завершаются.

У каждого потока есть собственная память для локальных переменных и доступ к общей памяти процесса.

Для определения, какие переменные считать глобальными, а какие локальными к директиве parallel добавляются параметры private() и shared() со списками имен соответствующих переменных. Есть так же параметр default, с помощью которого можно либо считать все переменные глобальными (значение shared), либо обязать прописывать private и shared вручную (значение none).

Если потоки обращаются к глобальной переменной, может возникнуть race condition (состояние гонки): если несколько потоков одновременно хотят изменить переменную, неизвестно в каком порядке они это сделают, и какой будет результат; если один поток хочет читать переменную, а другой изменять, неизвестно, старое или новое значение прочитает первый.

Для предотвращения этого существует директива critical, в которую оборачивается блок кода, взаимодействующий с глобальной переменной. Этот блок будет выполняться в каждый момент времени только одним потоком, а перед входом и после выхода из него синхронизировать со всеми потоками информацию об изменении переменной. Это очень сильно замедляет программу, поэтому в некоторых случаях (инкремент, декремент и x = x op y, где op одна из этих операций: +, *, -, /, &, ^, |, <<, >>) есть аналогичная оптимизированная директива atomic.

В определенных случаях типа суммирования, умножения, поиска максимума в цикле можно добавить параметр reduction(операция : переменная) в директиву parallel. Тогда у каждого потока создастся локальная копия глобальной переменной, а в конце параллельного участка ко всем этим копиям применится соответствующая операция, и результат запишется в глобальную переменную.

Для распараллеливания циклов есть директива omp for, которая часто совмещается с omp parallel (получается omp parallel for). Она распределяет итерации цикла между потоками в соответствии с параметром schedule.

- schedule(static) —итерации цикла будут поровну (приблизительно) поделены между потоками
- schedule(static, k) — блочно-циклическое распределение итераций. Каждый поток получает k итераций в начале цикла, затем (если остались итерации) процедура распределения продолжается. Планирование выполняется один раз, таким образом, каждый поток сразу узнает, какие итерации будет выполнять
- schedule(dynamic), schedule(dynamic, k) — динамическое планирование. По умолчанию параметр опции равен 1. Каждый поток

получает k итераций, выполняет их и запрашивает новую порцию среди еще нераспределенных. В отличие от статического планирования, выполняется многократно (во время выполнения программы). Конкретное распределение итераций между потоками зависит от темпов работы потоков и трудоемкости итераций

- schedule(guided), schedule(guided, k) — разновидность динамического планирования с изменяемым при каждом последующем распределении числе итераций. Распределение начинается с некоторого начального размера, зависящего от реализации библиотеки до значения, задаваемого в опции (по умолчанию 1). Размер выделяемой порции зависит от количества еще нераспределенных итераций

Описание алгоритма согласно варианту

Вариант: 9

Условие:

Автоматическая коррекция яркости изображения в цветовом пространстве YCbCr.601.

Значение пикселей изображения находится в диапазоне [0; 255]. Изображение может иметь плохую контрастность: используется не весь диапазон значений, а только его часть. Например, если самые тёмные места изображения имеют значение 20.

Задание состоит в том, чтобы изменить значения пикселей таким образом, чтобы получить максимальную контрастность (диапазон значений [0; 255]) и при этом не изменить цветность (оттенок). Этого можно достигнуть регулировкой контрастности в канале яркости Y цветового пространства YCbCr (601 в PC диапазоне: [0; 255]).

Важно: согласно стандарту PNM изображения хранятся в цветовом пространстве RGB.

<параметры_алгоритма> = <имя_входного_файла>

Входной файл содержит данные в формате PPM (P6).

В качестве выходного файла будет новое изображение в формате PPM (P6). Имя выходного файла для данного варианта гарантировано будет указано в аргументах.

Алгоритм:

Для каждого пикселя перевести RGB в YCbCr по формулам из спецификации, определить диапазон Y, для каждого пикселя пересчитать Y в диапазоне 0-255, затем перевести YCbCr в RGB по формулам из спецификации.

Практическая часть

Описание работы кода

1. Берем количество тредов из параметров запуска или дефолтное из omp_get_max_threads.
2. Читаем длину и ширину, представленные посимвольно в ASCII в ppm-файле.
3. Сохраняем информацию о каждом пикселе, значения r, g, b читаются уже побайтово. Это ввод, который невозможно распараллелить. Пиксели хранятся в одномерном массиве длины width*height, потому что итерирование по такому массиву параллелится лучше, чем по двумерному с помощью вложенных циклов.
4. Распараллеленно проходим по всем пикселям и переводим в YCbCr по спецификации jpeg, потому что она соответствует BT.601, но дает значения в диапазоне 0-255. При этом ищется минимальный и максимальный Y; чтобы не возникал race condition, используется параметр reduction.
5. Распараллеленно проходим по всем пикселям и переводим в RGB по спецификации jpeg. Для этого используется Y, пересчитанный по

формуле для перевода диапазона Ymin-Ymax в диапазон 0-255: (Y - Ymin) * 255 / (Ymax - Ymin).

6. Записываем в выходной файл заголовок ppm как строку и значения r, g, b каждого пикселя как байты. Это вывод, который невозможно распараллелить.

Графики

На графиках ниже бралось среднее за 10 запусков время работы в миллисекундах для картинки разрешением 1920x1080.

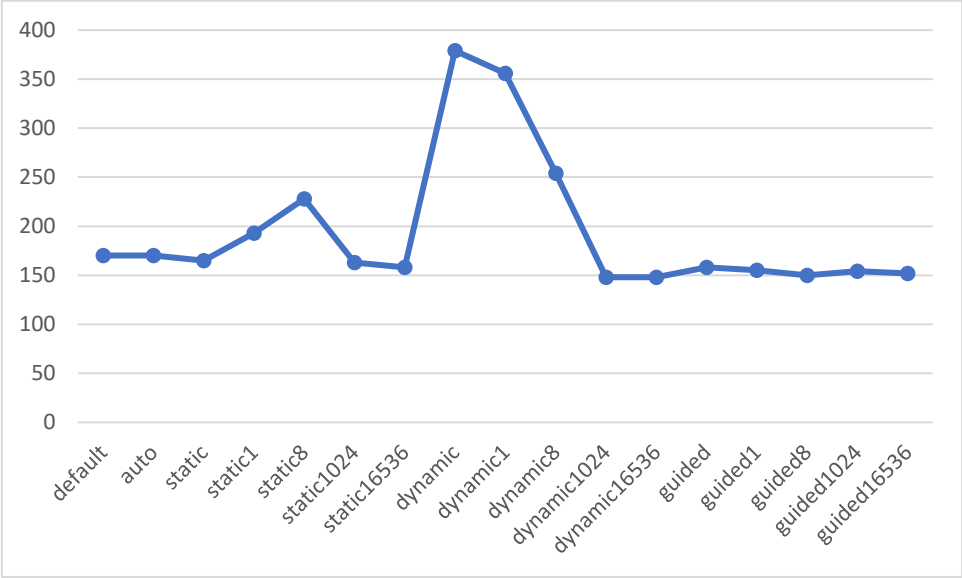

Рисунок №1 – Время работы при 6 потоках и различных значениях schedule

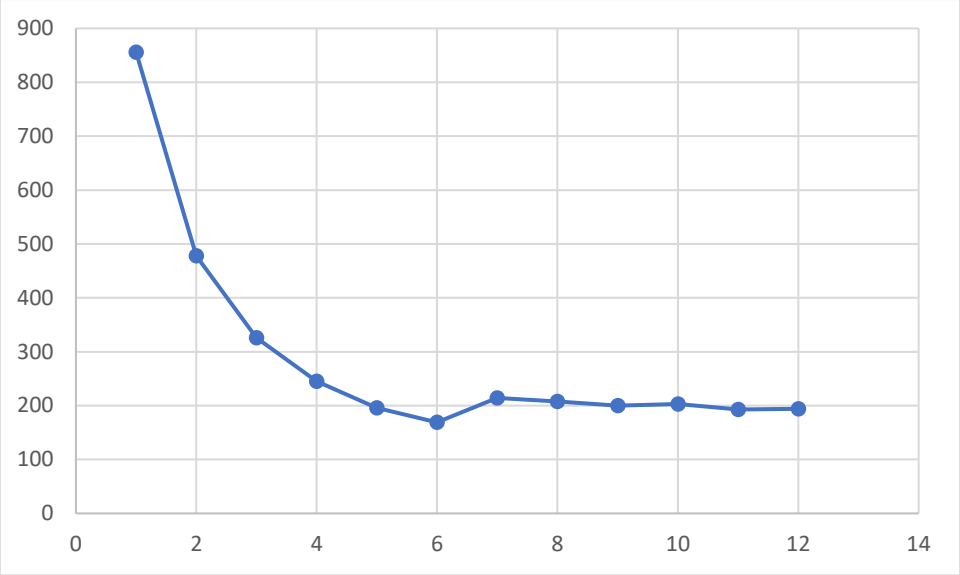

Рисунок №2 – Время работы при schedule(static) и различном количестве потоков

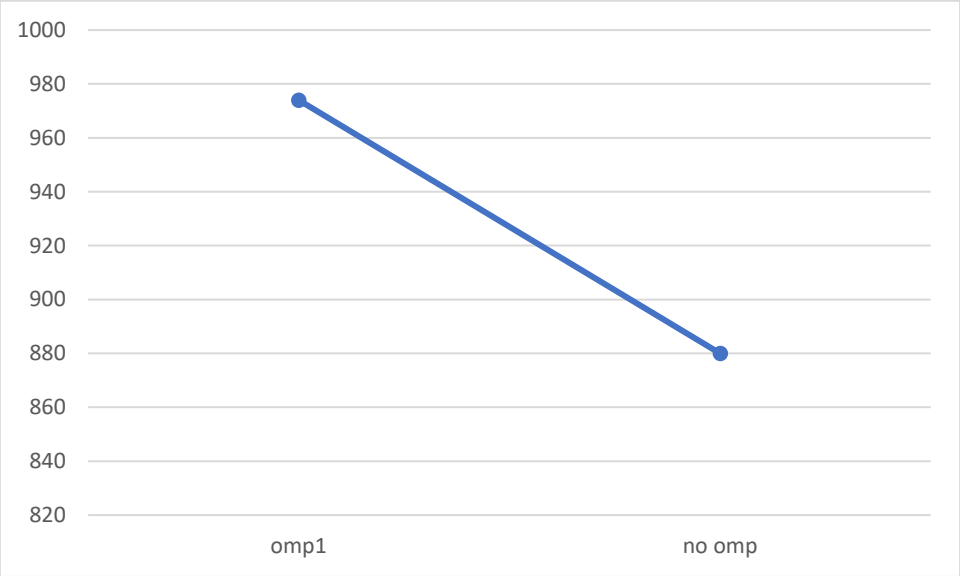

Рисунок №3 – Время работы с отключенным openmp и с одним потоком

Пример работы

Рисунок №4 – Исходная картинка с пониженной контрастностью

Рисунок №5 – Результат работы программы

Листинг

C++ 11, компилятор g++ с параметром запуска -fopenmp

main.cpp

```
#include <iostream>
#include <omp.h>
#include <vector>
#include <chrono>

typedef unsigned char uchar;
using namespace std;

vector<uchar> get_ycbcr(uchar r, uchar g, uchar b) {
    uchar y = 0.299 * r + 0.587 * g + 0.114 * b;
    uchar cb = 128 - 0.168736 * r - 0.331264 * g + 0.5 * b;
    uchar cr = 128 + 0.5 * r - 0.418688 * g - 0.081312 * b;

    return vector<uchar>{y, cb, cr};
}

uchar to_uchar(double d) {
    if (d < 0)
        return 0;
    if (d > 255)
        return 255;
    return uchar(d);
}

vector<uchar> get_rgb(uchar y, uchar cb, uchar cr) {
    uchar r = to_uchar(y + 1.402 * (cr - 128));
    uchar g = to_uchar(y - 0.34414 * (cb - 128) - 0.71414 * (cr -
128));
    uchar b = to_uchar(y + 1.772 * (cb - 128));

    return vector<uchar>{r, g, b};
}

int scale(int x, int mx, int mn) {
    if (mx == mn)
        return mx;
    return (x - mn) * 255 / (mx - mn);
}

int main(int argc, char *argv[]) {
    int threads = stoi(argv[1]);
    threads = (threads == 0) ? omp_get_max_threads() : threads;
```

```
    FILE *inf;
    if (!(inf = fopen(argv[2], "rb"))) {
        cout << "Can't open input file\n";
        return 0;
    }

    if (getc(inf) != 'P' || getc(inf) != '6') {
        cout << "Not ppm P6\n";
        return 0;
    }
    int w, h, mc;
    fscanf(inf, " %i %i %i ", &w, &h, &mc);

    vector<vector<uchar>> img(h * w);
    for (int i = 0; i < h * w; i++) {
        uchar r = getc(inf);
        uchar g = getc(inf);
        uchar b = getc(inf);
        img[i] = {r, g, b};
    }

    fclose(inf);

    // можно было бы переиспользовать img, если бы не нужно было
дублировать алгоритм без omp
    vector<vector<uchar>> p(h * w);

    // без omp
    uchar mx = 0, mn = 255;
    auto start = chrono::steady_clock::now();

    for (int i = 0; i < w * h; i++) {
        p[i] = get_ycbcr(img[i][0], img[i][1], img[i][2]);
        mx = max(mx, p[i][0]);
        mn = min(mn, p[i][0]);
    }

    for (int i = 0; i < h * w; i++) {
        p[i] = get_rgb(scale(p[i][0], mx, mn), p[i][1], p[i][2]);
    }

    auto end = chrono::steady_clock::now();
    printf("\nTime: %.3f ms\n", chrono::duration<float>(end -
start).count() * 1000);

    // c omp
    mx = 0, mn = 255;
    start = chrono::steady_clock::now();
```

```
#pragma omp parallel for schedule(static) reduction(max : mx)
reduction(min : mn) default(shared) num_threads(threads)
    for (int i = 0; i < w * h; i++) {
        p[i] = get_ycbcr(img[i][0], img[i][1], img[i][2]);
        mx = max(mx, p[i][0]);
        mn = min(mn, p[i][0]);
    }

#pragma omp parallel for schedule(static) default(shared)
num_threads(threads)
    for (int i = 0; i < h * w; i++) {
        p[i] = get_rgb(scale(p[i][0], mx, mn), p[i][1], p[i][2]);
    }

    end = chrono::steady_clock::now();
    printf("\nTime (%i thread(s)): %.3f ms\n", threads,
chrono::duration<float>(end - start).count() * 1000);


    FILE* outf;
    if (!(outf = fopen(argv[3], "wb"))) {
        cout << "Cant open output file\n";
        return 0;
    }

    fprintf(outf, "P6\n%d %d\n%d\n", w, h, 255);
    for (int i = 0; i < h * w; i++) {
        for (int k = 0; k < 3; k++) {
            fputc(p[i][k], outf);
        }
    }

    fclose(outf);

    return 0;
}
```