САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

«OPEN MP»

Выполнил: Леонтьев Тарас Михайлович

Номер ИСУ: 338916

студ. гр. М3135

Санкт-Петербург

2021

Цель работы: знакомство со стандартом Open MP.

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

Теоретическая часть 1

Gaussian blur using box blur approximation. Размытие по Гауссу – результат размытия изображения используя функцию Гаусса.

$$g(x,y) = e^{\frac{x^2+y\hat{}2}{2\sigma\hat{}2}}$$

Полученные значения вставляются в ядро свертки. Ядро свертки – это матрица, обычно размером $6\sigma - 1$, которая ходит по картинке и меняет каждый пиксель. σ – это стандартное отклонение цвета в изображении.

Но нам нужно не совсем Гауссовское размытие, а его аппроксимация, поскольку Гауссовское размытие работает достаточно медленно. В среде “Photoshop” используется аппроксимация Гауссовского размытия.

В аппроксимации размытия по Гауссу, мы прогоняем картинку через box blur 3 раза. Давайте определим horizontal и total blur.

$$b_h[i,j] = \sum_{x=j-br}^{j+br} \frac{f[i,x]}{2 \cdot br}$$

$$b_t[i,j] = \sum_{y=j-br}^{j+br} \frac{b_h[i,x]}{2 \cdot br}$$

Это все проходит цикл, и получается:

$$b_t[i,j] = \sum_{y=j-br}^{j+br} \frac{b_h[i,x]}{2 \cdot br} = \sum_{y=i-br}^{i+br} \sum_{x=j-br}^{j+br} \frac{f[y,x]}{(2 \cdot br)^2}$$

$$br = \sqrt{\frac{12 \cdot \sigma^2}{nb} + 1}$$

Теоретическая часть 2

OpenMP - это библиотека для параллельного программирования вычислительных систем с общей памятью. При этом используется параллелизм потоков. Потоки создаются в рамках единственного процессора и имеют свою память, также все потоки имеют доступ к памяти процессора.

Чтобы встретить параллельную область, надо написать "`#pragma omp parallel`". Процесс порождает ряд потоков, обычно столько, сколько позволяет компьютер, но можно и задавать количество потоков, обычно стоит задавать количество потоков не больше чем поддерживает компьютер, по скорости программа будет работать медленнее. Когда мы пишем "`#pragma omp parallel {. . .}`", (не вложенные в другие "`#pragma omp parallel {. . .}`"), потоки создаются, в этом случае, одним главным потоком, такой поток в OpenMP называется master. Но директивы parallel могут быть и вложенными, изменяется функцией (`omp_set_nested`). Стоит отметить, что все переменные, созданные до директивы parallel, являются общими для всех потоков, в противном случае, если они созданы внутри, то они являются локальными, при изменении общей переменной, непонятно, что будет, поэтому делать этого не стоит. Плохо еще читать переменную одним потоком, а другим ее изменять. Данные проблемы решаются директива critical. Правилом хорошего тона считается, если критическая секция содержит

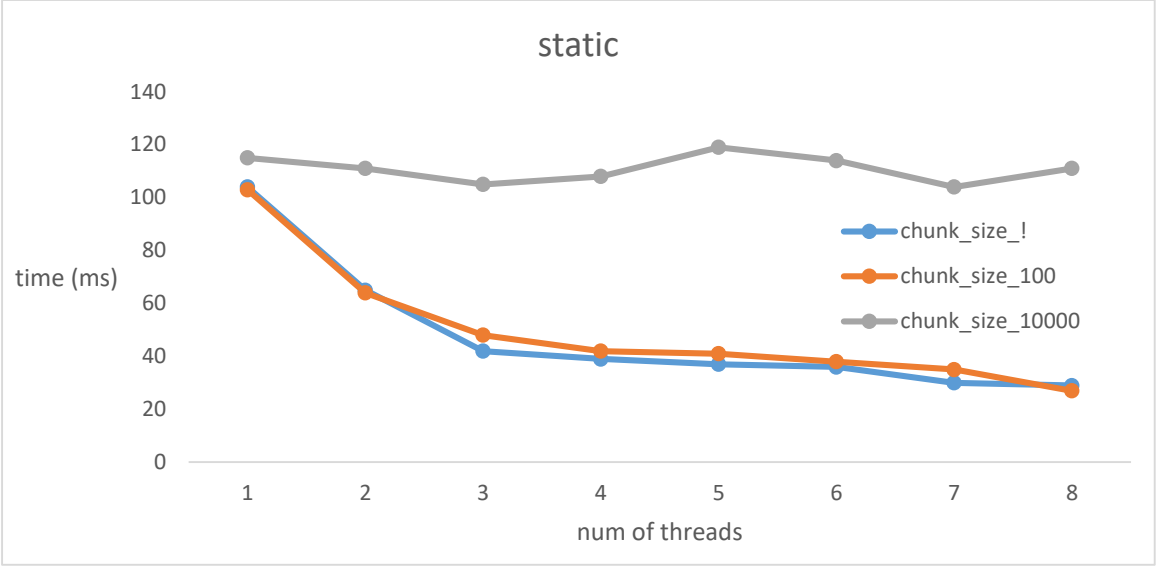
обращения только к одному разделяемому ресурсу. Но вообще, можно и использовать atomic, она ведет себя почти также, а работает быстрее, ее можно использовать для многих операций.

Самый популярный способ использовать OpenMP – это параллельный цикл. Параллельный цикл позволяет задать опцию schedule, изменяющую алгоритм распределения итераций между потоками, причем есть несколько опций планирования.

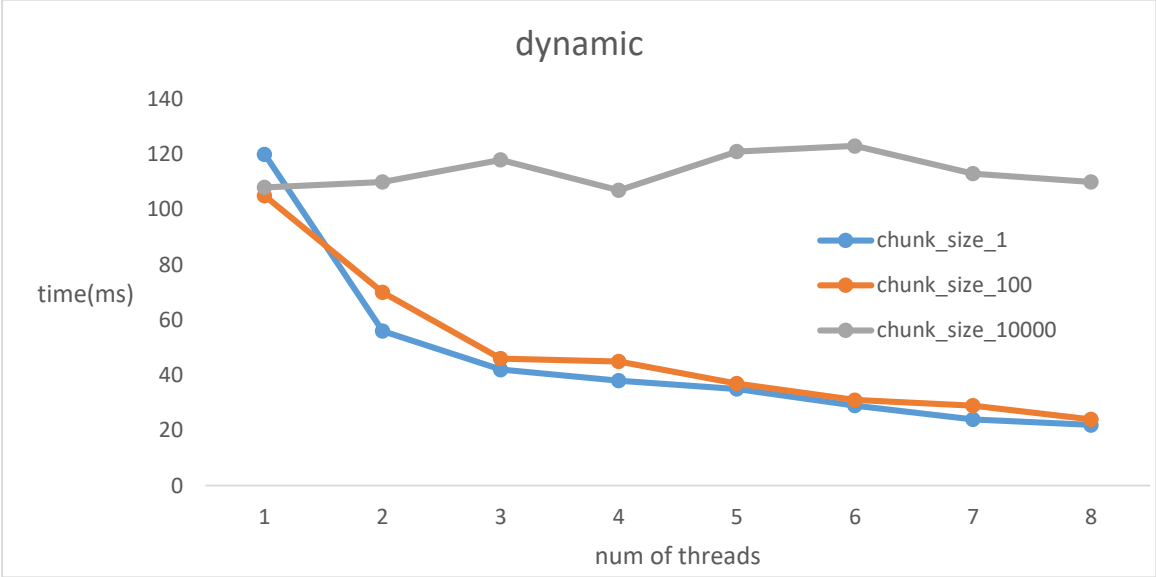
1. schedule(static) – статическое планирование. При его использовании итераций цикла будет поровну распределены между потоками.
2. schedule(static, x) – блочно-циклическое распределение итераций. Каждый поток получает заданное количество операций.
3. dynamic(static, x) – динамическое планирование. По умолчанию x равен 1. Каждый поток получает заданное количество итераций, выполняет их и просит сколько-то новых итераций. Также это происходит многократно, во время выполнения программы

Практическая часть

Я измерил время работы программы (аппроксимации размытия по Гауссу, с помощью box blur), с использованием разного планирования (static/dynamic) и с разным количеством тредов, также я менял внутри параметр chunk_size на 1, 100, 10000.



Время от количества тредов при планировании static график 1.

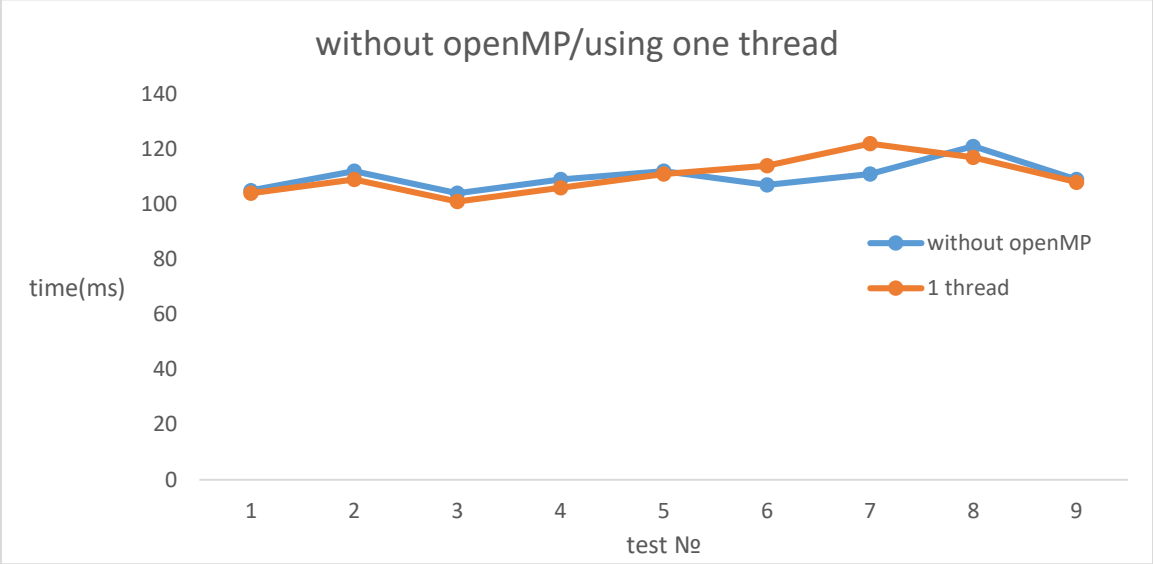


Время от количества тредов при планировании dynamic график 2.

Для получения данных результатов на каждом количестве тредов я брал среднее арифметическое из 5, чтобы было честно. Результатом наблюдение

есть то, что при увеличении количества тредов, но только до допустимой величины, той, которая поддерживается компьютер, время уменьшается. Если сделать количество тредов больше, чем сколько поддерживает компьютер программа будет работать медленнее.

Я еще зафиксировал, как работает программа без OpenMP и с одним тредом.



Без OpenMP vs 1 thread график 3.

Листинг

```
#include <vector>

#include <fstream>

#include <malloc.h>

#include <iostream>

#include <string>

#include <vector>

#include <istream>
```

```
#include <omp.h>
#include <chrono>
#include <cmath>

using namespace std;

typedef struct {
    string P5;
    int width;
    int height;
    int maxval;

    vector <vector <int>> pixel_map;

    vector <string> comments;
} pgm_file;

int get_box_radius(float sigma, int num_boxes) {
    return round(sqrt((12 * sigma * sigma / num_boxes) + 1));
}

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

    int num_threads = stoi(argv[1]);
```

```
    ifstream input_file;
    ofstream output_file;
    input_file.open(string(argv[2]));
    output_file.open(string(argv[3]), ios_base::binary);
    float sigma = stof(argv[4]);
    int num_boxes = stoi(argv[5]);

    pgm_file file;
    input_file >> file.P5;
    input_file >> file.width;
    input_file >> file.height;
    input_file >> file.maxval;

    file.pixel_map = vector <vector <int>>(file.height, vector
<int>(file.width));

    for (vector <int>& vect : file.pixel_map)
        for (int& pix : vect) {
            char temp = 0;
            input_file.get(temp);
            pix = (unsigned char) temp;
            if (file.maxval > 255) {
                input_file.get(temp);
                pix = (pix << 8) + (unsigned char) temp;
```

```
            }
        }


    vector <vector <int>> new_map = file.pixel_map;



    omp_set_num_threads(num_threads);
    int box_r = get_box_radius(sigma, num_boxes);

    auto t1 = chrono::high_resolution_clock::now();

    for (int b = 0; b < 3; b++) {
        //Horizontal box approximation
#pragma omp parallel for schedule(static, 100)
        for (int i = 0; i < file.height; i++) {
            for (int j = 0; j < file.width; j++) {
                double kernel = 0;
                for (int x = max(j - box_r, 0); x < min(j + box_r + 1,
(int)file.width); x++)
                    kernel += file.pixel_map[i][x];
                kernel /= (box_r + box_r + 1);
                new_map[i][j] = round(kernel);
            }
```

```
        }

        file.pixel_map = new_map;

        //vertical box approximation

#pragma omp parallel for schedule(static, 100)
        for (int i =0; i < file.height; i++) {
            for (int j = 0; j < file.width; j++) {
                double kernel = 0;
                for (int y = max(i - box_r, 0); y < min(i + box_r + 1,
(int)file.height); y++)
                    kernel += file.pixel_map[y][j];
                kernel /= (box_r + box_r + 1);
                new_map[i][j] = round(kernel);
            }
        }
    }

    auto t2 = chrono::high_resolution_clock::now();
    auto time = chrono::duration_cast <chrono::milliseconds> (t2 - t1);
    cout << "Vertical/Horizontal Box approxiamtion algorithm runtime: " <<
time.count() << "ms\n";
    file.pixel_map = new_map;
```

```
    output_file.write(file.P5.c_str(), 2);
    output_file.write(" ", 1);
    output_file.write(to_string(file.width).c_str(),
to_string(file.width).size());
    output_file.write(" ", 1);
    output_file.write(to_string(file.height).c_str(),
to_string(file.height).size());
    output_file.write(" ", 1);
    output_file.write(to_string(file.maxval).c_str(),
to_string(file.maxval).size());
    output_file.write(" ", 1);



    int i, j;
    for (i = 0; i < file.height; i++)
        for (j = 0; j < file.width; j++) {
            if (file.maxval > 255) {
                output_file.write((char*)&new_map[i][j], 2);
            }
            else {
                uint8_t temp = new_map[i][j];
                output_file << temp;
            }
        }
```

```
    input_file.close();

    output_file.close();

    return time.count();

}
```