САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчёт

по домашней работе №5:

OpenMp

Выполнил: Рожков Денис Валерьевич

Номер ИСУ: 334899

студ. гр. М3134

Санкт-Петербург

2022

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.

Описание принципов работы ключевых элементов OpenMP.

В своем отчете я опишу только те элементы стандарта OpenMP, которые понадобились мне при написании и оптимизации программы. Сразу обозначу, я выполнял модификацию Hard. Вступительная часть на этом закончена, перейдем к теории.

Принцип работы

OpenMP - это интерфейс прикладного программирования для создания многопоточных приложений, предназначенных в основном для параллельных вычислительных систем с общей памятью.

Данный интерфейс позволяет создавать многопоточные приложения на языках программирования Fortran и C/C++, с помощью распределения тредов(потоков) по разным ядрам вычислительной машины. Так что, эффективность данной идеи многом зависит еще и от производительности компьютера.

Ключевые элементы.

Основные конструкции OpenMP - это директивы компилятора или прагмы (директивы препроцессора) языка C/C++ или Fortran.

Конструктивно в составе технологии OpenMP можно выделить:

- Директив;

- Библиотеку функций;

- Набор переменных окружения.

В своем отчете я рассмотрю только то, что мне пригодилось из стандарта OpenMP.

В самом общем виде формат директив OpenMP может быть представлен в следующем виде:

```
#pragma omp  <имя директивы> [<параметр>, …, <параметр>].
```

Начальная часть директивы (#pragma omp) является фиксировауной, вид директивы определяется ее именем (имя_директивы), каждая директива может сопровождаться произвольным количеством параметров.

Конструкция parallel – самая часто используемая директива OpenMP. Она предназначена для того, чтобы создать доп поток для следующего структурированного блока(более подробно ниже).

Особенности :

1) когда основной поток достигает директивы parallel он порождает определенное количество тредов. Основному потоку присваивается индекс 0. Количество тредов может задаваться вручную несколькими методами.

Пример методами, которым я пользовался:

```
{
#pragma omp parallel num_threads(num_thread)
//какой либо код
}
```

По условию задачи количество потоков задается програмно, я данное значение обозначил переменной num_thread. С помощью такой простой строки кода, будет распараллелена та часть кода которая идет в данном логическом блоке на количество тредов в точности равное num_thread.

2) В конце программного блока директивы обеспечивается синхронизация потоков – выполняется ожидание окончания вычислений всех потоков; далее все потоки завершаются – дальнейшие вычисления продолжает выполнять только основной поток.

Условия выполнения:

Условия выполнения определяют то, как будет выполняться параллельный участок кода и область видимости переменных внутри этого участка кода.

При выполнении задания мне понадобилось 3 условия, вот они:

1)shared(var1, var2, ...) Условие shared указывает на то, что все перечисленные переменные будут разделяться между потоками. Все потоки будут доступаться к одной и той же области памяти.

2) private(var1, var2, ...) Условие private указывает на то, что каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.

3) reduction(оператор:var1, var2, ...) Это условие гарантирует безопасное выполнение операций редукции, например, вычисление глобальной суммы.

Это условие позволяет производить безопасное глобальное вычисление. Приватная копия каждой перечисленной переменной инициализируется при входе в параллельную секцию в соответствии с указанным оператором (0 для оператора +). При выходе из параллельной секции из частично вычисленных значений вычисляется результирующее и передается в основной поток.

Параллельный цикл for:

Цель конструкции – распределение итераций цикла по потокам. Пример:

```
#pragma omp parallel
  {
    #pragma omp for private(i) shared(a,b)
        for(i=0; i < size; ++i)
            a[i] = a[i] + b[i];
   }
```

По умолчанию флагом для окончания выполнения потоков является конец цикла. Каждый поток достигнув конца дожидается последнего. После чего основной поток продолжает свое действие. Так же есть условие no_wait, которое позволяет не ждать оставшиеся потоки.

4) Условие schedule.

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

schedule(тип , размер блока).

Размер блока задает количство итераций в каждом треде. Тип расписания может принимать следующие значение:

- static – итерации равномерно распределяются по потокам. К примеру если в цикле 100 итераций и 5 потоков, то на каждый будет распределено по 20 итераций. Пронумерованных последовательно.

- dynamic – работа распределяется пакетами заданного размера (по умолчанию размер равен 1) между потоками. Как только какой-либо из потоков заканчивает обработку своего пакета, он начинает обрабатывать следующую.

Другие возможные значение schedule я не использовал.

На этом вещи, которые я использовал из стандарта openMP закончились.

Описание работы, написанного кода.

1) Само собой изначально необходимо считать данные и проверить их на корректность. За этот пункт в моей программе отвечает процедура readFile. Данные я храню в одномерном массиве pixel.

2) Далее необходимо найти минимальный и максимальный пиксель в каждом из каналом(в .pgm он один). Это я делаю с помощью цифровой сортировки, которая легко даст мне понять какие пиксели мне

необходимы, после того как я пропущу kf * 100 % пикселей сверху и снизу(см. функцию get_minmax).

3) Далее находим самый минимальный и максимальный полученные пискели соответственно по каждому каналу и в соответствии с ними меняем каждое значение(именно значение) в соответствии с формулой(см. get_pixel).

4)Выводим время, за которое работает программа и итоговый файл.

Экспериментальная часть

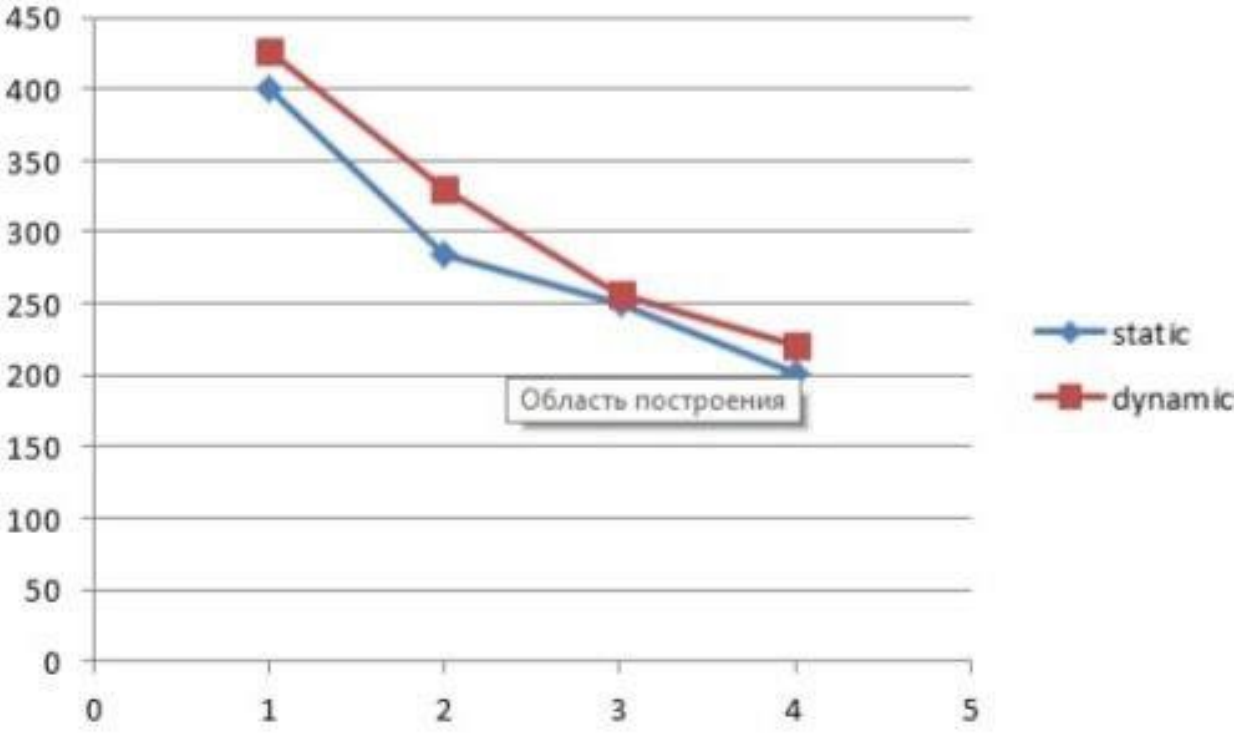

Рисунок №1 – разное кол-во потоков.

Вертикально расположено время(мс), горизонтально кол-во потоков.

У меня на ноутбуке 4 ядра.

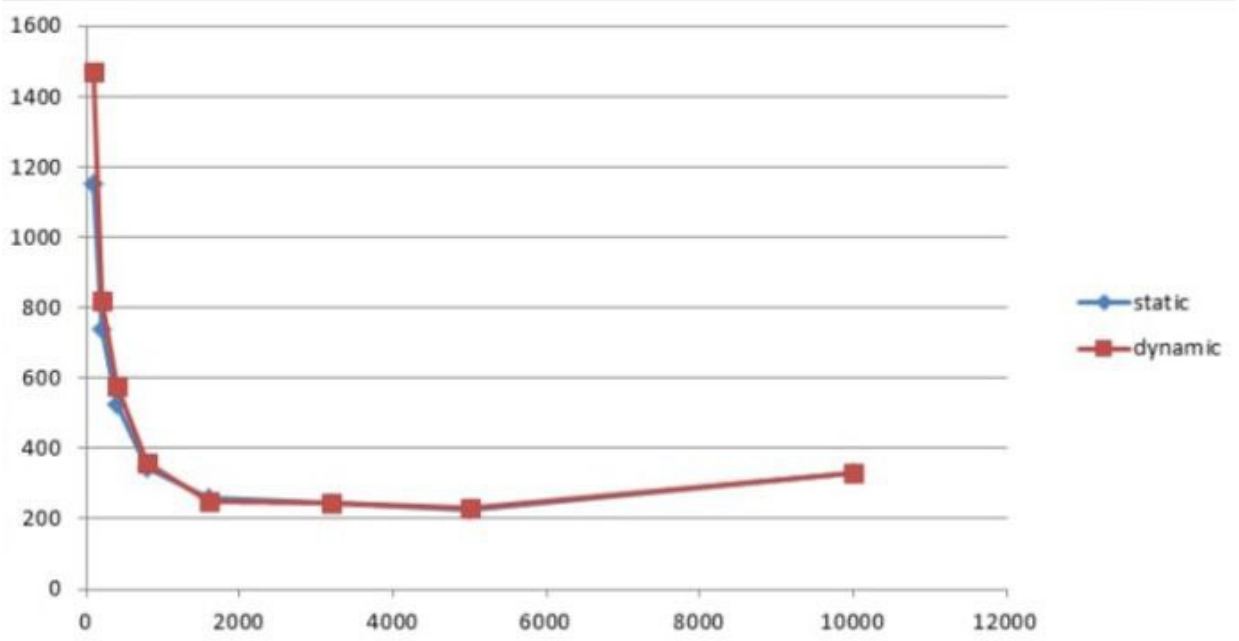

Рисунок №2 – одинаковое кол-во потоков, разный параметр schedule.

Вертикально расположено время(мс), горизонтально – размер schedule.

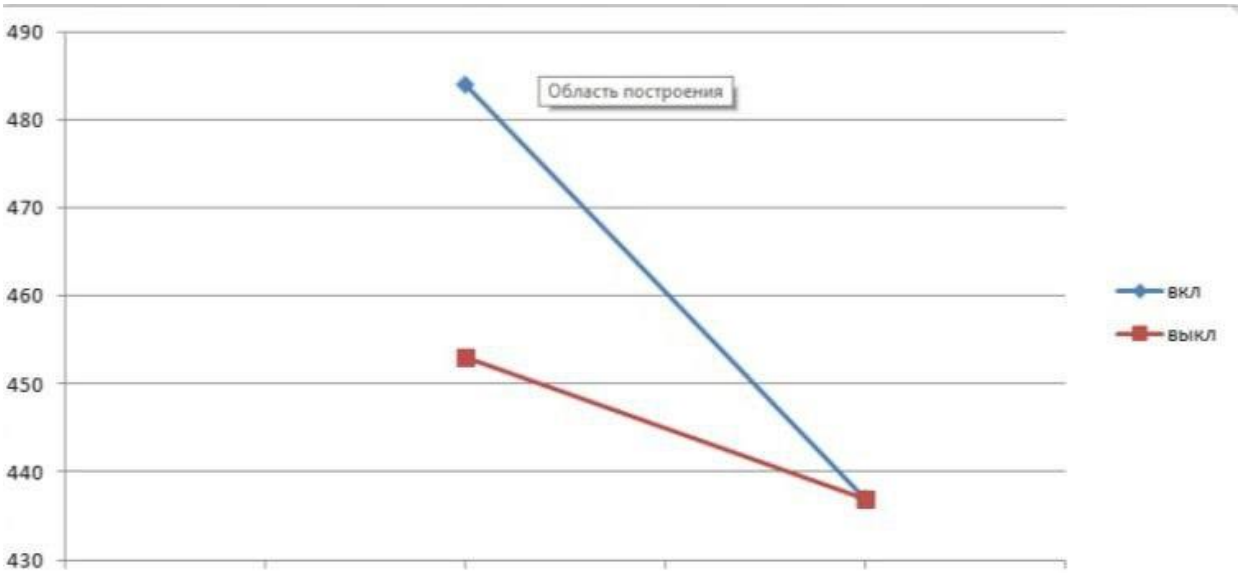

Рисунок №3 – 1 поток и ВКЛ/ВЫКЛ OpenMP.

Вертикально расположено время(мс), 1ые 2 точки слева описывают включенное OpenMp, а вторые выключенное.

Листинг кода:

main.cpp

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <cmath>
#include <chrono>
#include <thread>
#include <omp.h>
typedef unsigned char uchar;
//4 pictest7.ppm res.ppm 0.02
using namespace std;

struct node {
    float mn, mx;
};
int num_thread, block;
float kf;

int readFile(int &argc, char *argv[], int &type, int &w, int &h, int &dummy, uchar **pixel) {
    if (argc != 5) {
        cerr << "invalid number of arguments";
        return 2;
```

```
    }
    FILE *f_in;
    fopen_s(&f_in,argv[2],"rb");
    if (f_in == NULL) {
        cerr << "Invalid input file";
        fclose(f_in);
        return 3;
    }
    fscanf_s(f_in,"P%i%i%i\n",&type, &w, &h, &dummy);
    if ((type < 5 || type > 6) || (w <= 0 || h <= 0) ||
(dummy != 255)) {
        cerr << "Incorrect parameters";
        fclose(f_in);
        return 4;
    }

    if (type == 5) {
        *pixel = (uchar *) malloc(sizeof(uchar) * h * w);
    } else {
        *pixel = (uchar *) malloc(sizeof(uchar) * h * w *
3);
    }
    if (*pixel == NULL) {
        cerr << "Allocation memory failed";
```

```
    fclose(f_in);
    return 5;
}
int startPosition = ftell(f_in);
fseek(f_in, 0, SEEK_END);
int countPixels = ftell(f_in) - startPosition;
fseek(f_in, startPosition, 0);
if (type == 5) {
    if (countPixels != w * h) {
        cerr << "Not enough data";
        fclose(f_in);
        return 6;
    }
    fread(*pixel, sizeof(uchar), w * h, f_in);
} else {
    if (countPixels != w * h * 3) {
        cerr << "Not enough data";
        fclose(f_in);
        return 6;
    }
    fread(*pixel, sizeof(uchar), w * h * 3, f_in);
}
fclose(f_in);
return 0;
```

```
}

int get_pixel(node &x, unsigned char &p) {
    if (x.mx == x.mn) {
        return p;
    }
    float pix = (p - x.mn) / (x.mx - x.mn);
    if (pix >= 1.0) {
        return 255;
    }
    if (pix <= 0.0) {
        return 0;
    }
    return pix * 255;
}

node get_minmax(int *cnt, int flag) {
    int sum = 0;
    int mn = 0, mx = 0;
    for (int i = 0; i < 256; ++i) {
        sum += cnt[i];
        if (sum > flag) {
            mn = i;
            break;
```

```
        }
    }
    sum = 0;
    for (int i = 255; i >= 0; --i) {
        sum += cnt[i];
        if (sum > flag) {
            mx = i;
            break;
        }
    }
    return {mn + 0.0f, mx + 0.0f};
}

void go_pgm(uchar *pixel, int &size) {
    int cnt[256] = {0};
    {
#pragma omp parallel num_threads(num_thread) shared(cnt, size)
        {
#pragma omp for schedule(static, block)
            for (int i = 0; i < size; ++i) {
                ++cnt[(pixel)[i]];
            }
        }
```

```
    }
    node x = get_minmax(cnt, round(size * kf));
    {
#pragma omp parallel num_threads(num_thread) shared(size)
        {
#pragma omp for schedule(static, block)
            for (int i = 0; i < size; ++i) {
                (pixel)[i] = get_pixel(x, (pixel)[i]);
            }
        }
    }
}

void go_ppm(uchar *pixel, int &size) {
    int r[256] = {0}, g[256] = {0}, b[256] = {0};
    omp_set_num_threads(num_thread);
#pragma omp parallel for schedule(static, block) reduction(+:r[:], g[:], b[:]) shared(pixel, size)
    for (size_t i = 0; i < size; i += 3) {
        ++r[(pixel)[i]];
        ++g[(pixel)[i + 1]];
        ++b[(pixel)[i + 2]];
    }
    int flag = (size / 3) * kf;
```

```
    node rr = get_minmax(r, flag);
    node gg = get_minmax(g, flag);
    node bb = get_minmax(b, flag);
    node x = {min(min(rr.mn, gg.mn), bb.mn), max(max(rr.mx, gg.mx), bb.mx)};
    omp_set_num_threads(num_thread);
#pragma omp parallel for schedule(static, block) shared(x, size)
    for (size_t i = 0; i < size; ++i) {
        (pixel)[i] = get_pixel(x, (pixel)[i]);
    }
}

int main(int argc, char *argv[]) {
    int type, w, h, dummy;
    uchar *pixel;
    int result = readFile(argc, argv, type, w, h, dummy, &pixel);
    if (result == 0) {
        char *fileName_out = argv[3];
        num_thread = stoi(argv[1]);
        kf = stof(argv[4]);

        if (kf < 0 || kf >= 0.5) {
            cerr << "Invalid kf";
```

```
    free(pixel);
    return 7;
}
if (fileName_out == NULL) {
    cerr << "Invalid output file";
    free(pixel);
    return 8;
}
FILE *f_out;
fopen_s(&f_out, fileName_out, "wb");
if (f_out == NULL) {
    cerr << "Allocation memory failed";
    free(pixel);
    return 9;
}
int size = w * h;
block = 4096;
unsigned int start_time = clock();
if (type == 5) {
    go_pgm(&*pixel, size);
} else if (type == 6) {
    size *= 3;
    go_ppm(&*pixel, size);
}
```

```
        unsigned int end_time = clock();
        cout << end_time - start_time << "mc" << endl;
        fprintf(f_out,  "P%i\n%i  %i\n%i\n",  type,  w,  h,
dummy);
        fwrite(pixel, sizeof(uchar), size, f_out);
        free(pixel);
        fclose(f_out);
        cout << "Successful" << endl;
        return 0;
    } else {
        return result;
    }
}
```