САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил: Султанов Мирзомансурхон Махсудович

Номер ИСУ: 311629

студ. гр. M3134

Санкт-Петербург

2022

Цель работы: знакомство со стандартом OpenMP.

Инструментарий и требования к работе: C++. Стандарт OpenMP 2.0.

Теоретическая часть

OpenMP помогает писать программы, которые не требуют сильных изменений, чтобы перевести программу в параллельный режим. В целом, OpenMP – это открытый стандарт для написания параллельных программ для систем с общей памятью. Официально поддерживаются C, C++ и Fortran. Несмотря на это, можно найти реализации и для некоторых других языков, к примеру, Java. OpenMP даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Существует множество разновидностей параллельных вычислительных систем – многоядерные/многопроцессорные компьютеры, кластеры, системы на видеокартах и др. Библиотека OpenMP подходит только для программирования систем с общей памятью, где при этом используется параллелизм потоков. Потоки же создаются в рамках единственного процесса и имеют собственную память. Кроме того, все потоки имеют доступ к памяти процесса (см. рисунок 1).

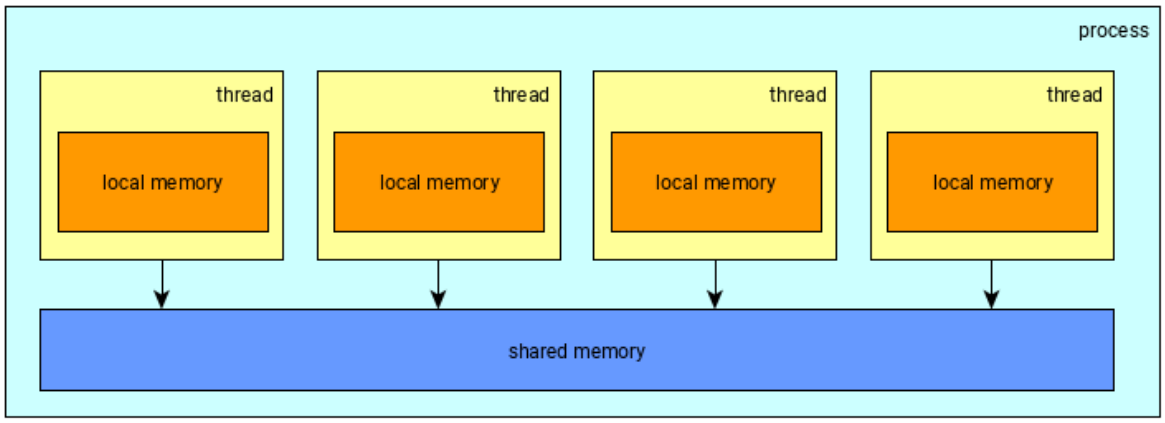


Рисунок 1 – модель памяти в OpenMP

В модели с разделяемой памятью взаимодействие потоков происходит через локальные переменные, доступ к которым имеет лишь соответствующий поток. При неправильном обращении с такими переменными в программе могут возникнуть ошибки соревнования (race condition). Ошибка возникает вследствие того, что потоки выполняются параллельно и соответственно последовательность доступа к разделяемым переменным может быть различна от одного запуска программы к другому. Для контроля ошибок соревнования работу потоков необходимо синхронизировать. Для разных же программ нужно по-разному синхронизировать переменные. Отчасти именно поэтому параллельное программирование — это бремя программистов, и оно не имеет аппаратного решения. Для решения этой проблемы используются такие примитивы синхронизации как критические секции, барьеры, атомарные операции и блокировки.

Количество задаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне при помощи переменных окружения.

Для того, чтобы программа, написанная на C++, могла использовать библиотеки OpenMP, необходимо подключить заголовочный файл “omp.h”, а также во время компиляции через командную строку добавить опцию сборки –fopenmp для компиляторов gcc и g++ или установить соответствующий флаг в настройках проекта (для Visual Studio или CLion). Для некоторых компиляторов иногда требуется докачать дополнительные библиотеки, чтобы опция –fopenmp работала корректно.

В целом, ключевыми элементами OpenMP являются: конструкции для создания потоков (директива parallel); конструкции распределения работы между потоками (директивы DO/for и section); конструкции для управления работой с данными (выражения shared и private для определения класса памяти переменных); конструкции для синхронизации потоков

(директивы critical, atomic и barrier); процедуры библиотеки поддержки времени выполнения (например, omp_get_thread_num); переменные окружения (например, OMP_NUM_THREADS).

После запуска программы создаётся единственный процесс, который начинает выполняться, как и обычная последовательная программа. Встретив параллельную область, задаваемую #pragma omp parallel, процесс порождает ряд потоков (их число можно задать явно, однако по умолчанию будет создано столько потоков, сколько в имеющейся системе вычислительных ядер). Для того чтобы указать количество потоков, нужно использовать функцию из библиотеки <omp.h>: omp_set_num_threads(num), где num – количество потоков. Границы параллельной области выделяются фигурными скобками, в конце области потоки уничтожаются. Схематично этот процесс изображён на 2 рисунке.

```
#include "omp.h"

int main() {
  // A - single thread
#pragma omp parallel
  {
    // B - many threads
  }
  // C - single thread
#pragma omp parallel
  {
    // D - many threads
  }
  // E - single thread
}
```

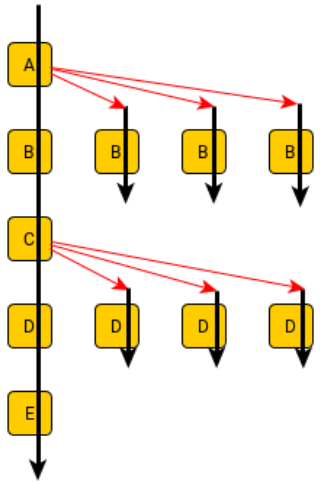


Рисунок 2 – Директива omp parallel

Чёрными линии на рисунке – это время жизни потоков, а красные – этот момент порождения. Видно, что все потоки создаются одним (главным) потоком, который существует всё время работы процесса. Такой поток в OpenMP называется master, все остальные потоки многократно создаются и уничтожаются. Стоит отметить, что директивы parallel могут

быть вложенными, при этом в зависимости от настроек могут создаваться вложенные потоки.

Директива for используется для явного распараллеливания следующего цикла for, при этом каждая нить начинается со своего индекса. Если не указывать директиву, то цикл будет пройден каждой нитью полностью от начала и до конца.

Параметр schedule для директив с циклом (для таких, как for) нужен для планирования распределения итераций цикла между потоками. Существует пять различных типов параметра schedule: static, dynamic, guided, auto, runtime.

Для static вся совокупность загружаемых процессов разбивается на равные порции размера chunk, и эти порции последовательно распределяются между процессорами потоками с первого до последнего. Если chunk отсутствует, то OpenMP разделяет итерации на фрагменты примерно равного размера и распределяет не более одного фрагмента на каждый поток. Примеры показаны на рисунке 3.

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************

schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****

schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

Рисунок 3 – Примеры работы static schedule

Для dynamic вся совокупность загружаемых процессов, как и в предыдущем варианте, разбивается на равные порции размера chunk, но эти порции загружаются последовательно в освободившиеся потоки (процессоры). Если отсутствует chunk, вызывается schedule(dynamic, 1). Примеры показаны на рисунке 4.

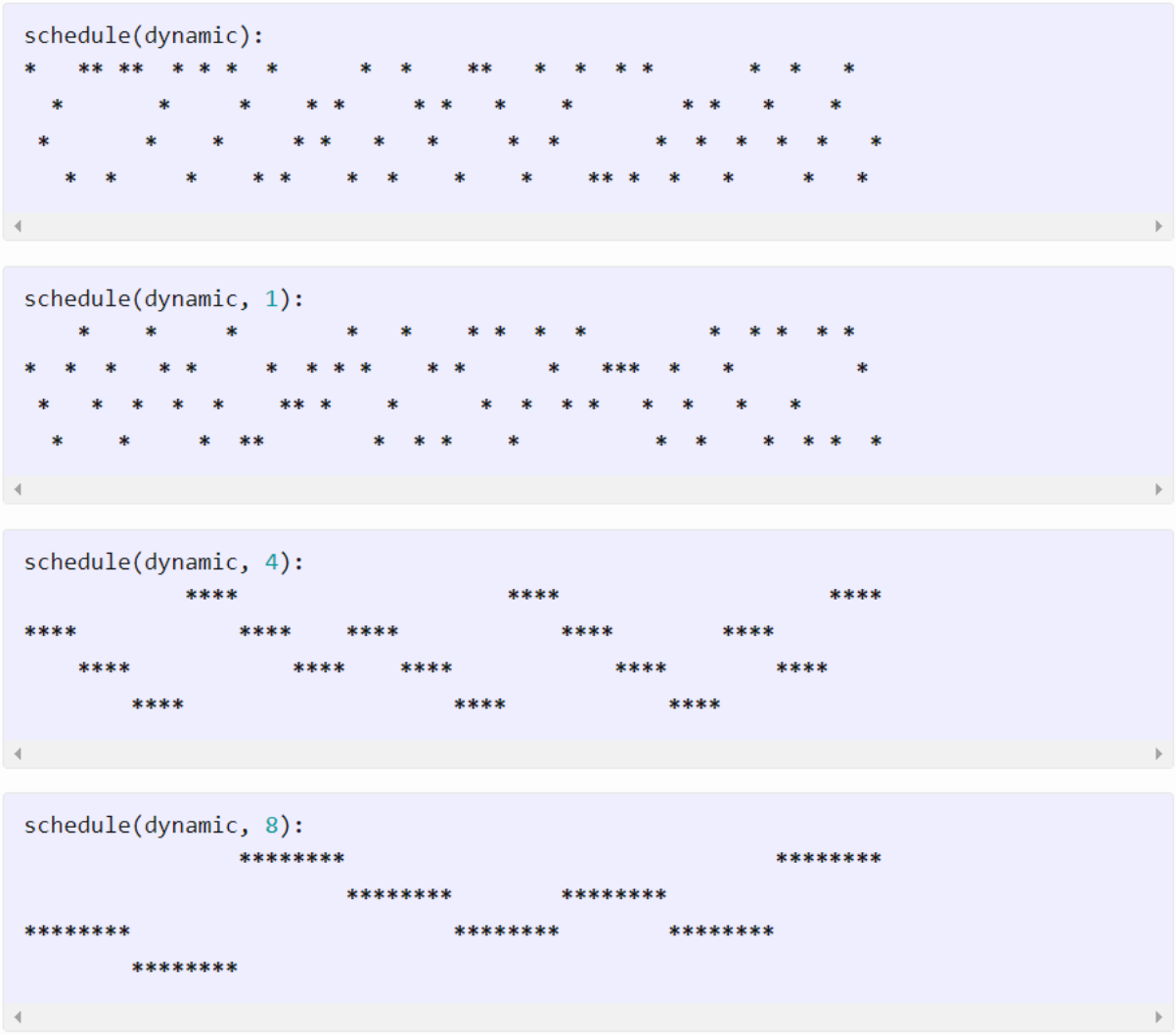


Рисунок 4 – Примеры работы dynamic schedule

Guided аналогичен dynamic. OpenMP снова делит итерации на части. Каждый поток выполняет часть итераций, а затем запрашивает другой фрагмент, пока не закончатся доступные фрагменты. Разница с типом динамического программирования заключается в размере блоков. Размер чанка пропорционален количеству неназначенных итераций, разделённому на количество потоков. Поэтому размер кусков уменьшается.

Минимальный размер чанка устанавливает значение chunk, однако фрагмент, содержащий последнюю итерацию, может иметь размер меньше, чем chunk. Если отсутствует chunk, вызывается schedule(guided, 1) Примеры показаны на рисунке 5.

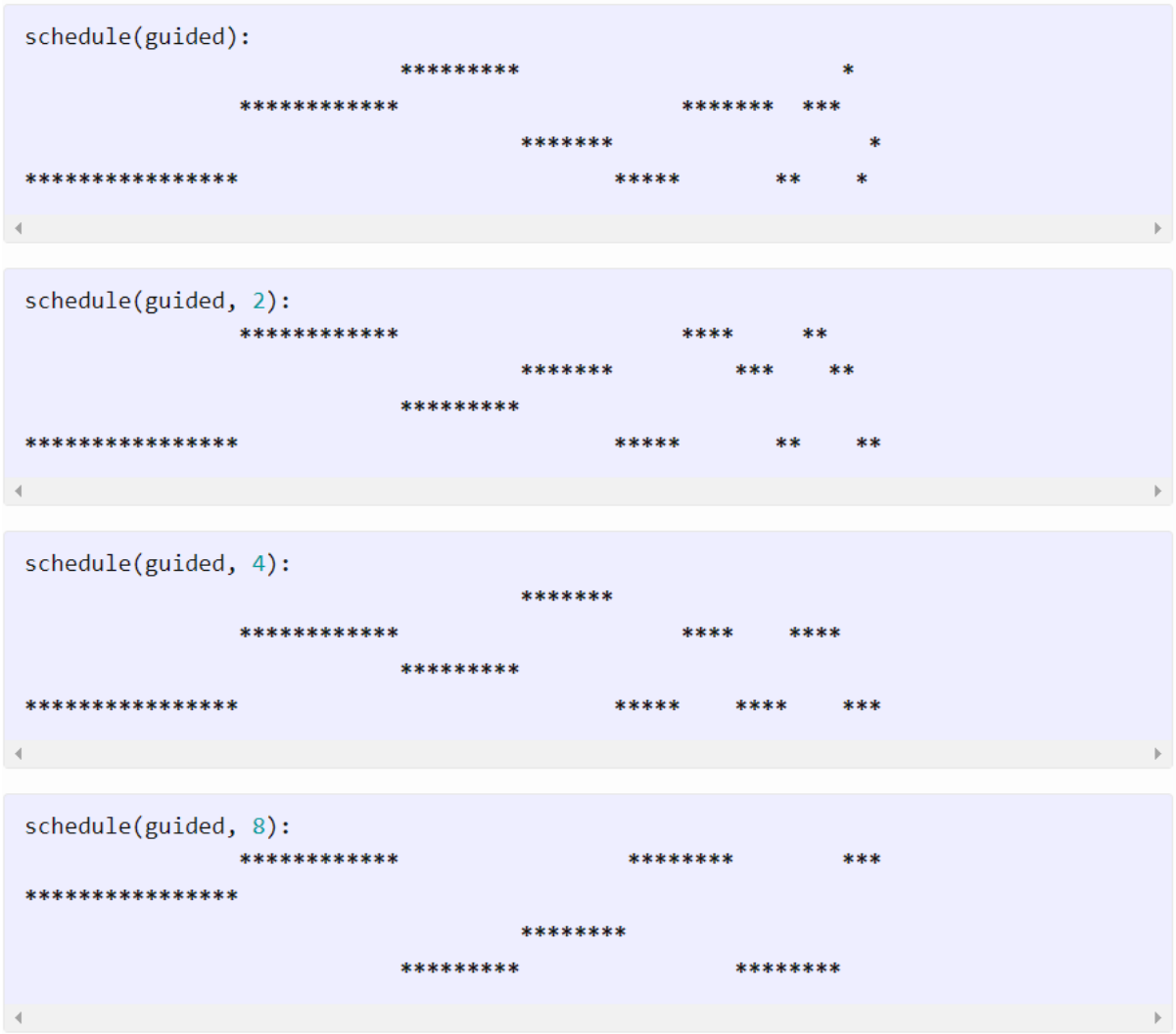


Рисунок 5 – примеры работы guided schedule

Auto – тип, при котором компилятор или исполняющая система сами выбирают один из трёх ранее рассказанных типов по своему усмотрению.

Runtime – этот тип выбирает один из трёх первых типов, который указан в переменной окружения OMP_SCHEDULE. Это полезно, если в зависимости от ситуации необходимо выбрать какой-то конкретный тип schedule.

Теперь перейдём к устройству директив atomic и reduction.

Идея использования atomic заключается в следующем. Для каждого потока у нас есть свой локальный счётчик. А глобальный счётчик будет помечен как atomic, т.е. увеличивать этот счётчик может лишь один поток в одно время. Для остальных же потоков доступ к глобальному счётчику закрыт, и они будут ждать, когда процесс, работающий с глобальным счётчиком, завершится.

Принцип работы директивы reduction выглядит так. Сначала для каждой переменной создаются локальные копии в каждом потоке. Локальные копии инициализируются соответственно типу оператора. Для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или его аналоги. Все исходные значения переменных можно увидеть на рисунке 6. И наконец, над локальными копиями переменных после выполнения операторов параллельной области выполняется заданный оператор. Порядок выполнения не определён.

Оператор	Исходное значение переменной
+	0
*	1
-	0
&	~0 (каждый бит установлен)
|	0
^	0
&&	1
||	0

Рисунок 6 – Операторы reduction и их исходные значения переменных

Описание работы написанного кода (Hard)

В целом, работу кода, помимо чтения и записи данных, можно поделить на два этапа:

1) Построение вспомогательного массива для подсчёта количества встречаемости каждого байта и нахождение границ исходного диапазона. Это нужно для нахождения исходного диапазона с учётом коэффициента игнора. С помощью коэффициента игнора и количества байтов находим нужное количество байтов для игнора. И находим минимальный префикс и суффикс, сумма которого будет больше нужного количества байтов. С помощью этого мы сможем найти границы исходного диапазона – Amin и Amax. При этом для RGB каналов мы находим отдельно минимумы и максимумы и берём минимум из минимумов для каждого канала и максимум из максимумов для каждого канала. Таким образом, мы одинаково сможем изменять каналы R, G, B.

2) Теперь надо увеличить собственно контрастность изображения. Это можно сделать по формуле a = (a – Amin) * 255/(Amax – Amin), что соответствует увеличению разницы между значениями каждого пикселя и растяжению до диапазона [0…255], где a – это произвольный байт изображения.

Теперь о том, как распараллеливать это чудо. Замечу, что во время работы кода программа проходит через 3 больших цикла (опять таки не считая чтение и запись): подсчёт байтиков, нахождение минимума и максимума и изменение каждого байта. При этом нахождение минимума или максимума занимает максимум 256 итераций (на самом деле обычно гораздо меньше), поэтому нет большого смысла распараллеливать это цикл. А вот подсчёт байтов и изменение каждого байта происходит для каждого байта зависит от размера файла, поэтому занимает много времени.

Для распараллеливания подсчёта байтиков надо использовать либо atomic, либо reduction, потому что возможно обращение к одинаковым участкам памяти в разных потоках. В целом, reduction показывает себя

лучше в данном случае, поэтому дальнейшие вычисления идут с учётом того, что распараллеливание происходило с директивой reduction.

Для распараллеливания изменения байтов обращения к одинаковым участкам памяти в разных потоках невозможно, поэтому достаточно просто включить #pragma omp for с нужным режимом и количеством чанков.

Говоря, о чанках, в моём коде я использовал 512. Для чего это? Нет никакого смысла в распараллеливании, если в рамках одного потока часто происходят кэш-промахи, потому что в таком случае мы будем больше тратить время на обращение к памяти нежели на вычисления, что может привести даже к результату хуже нежели без распараллеливания. 512 вполне достаточно для избегания данной ситуации. Однако на маленьких файлах из-за этого возможно, что не получится работать с большим количеством потоков эффективно, так как некоторые потоки могут быть не заполнены данными.

Ещё немного слов про компиляцию программы. Компиляция происходила следующим образом:

g++ -fopenmp -O0 -o mainpnm.exe .\pnmParser.cpp

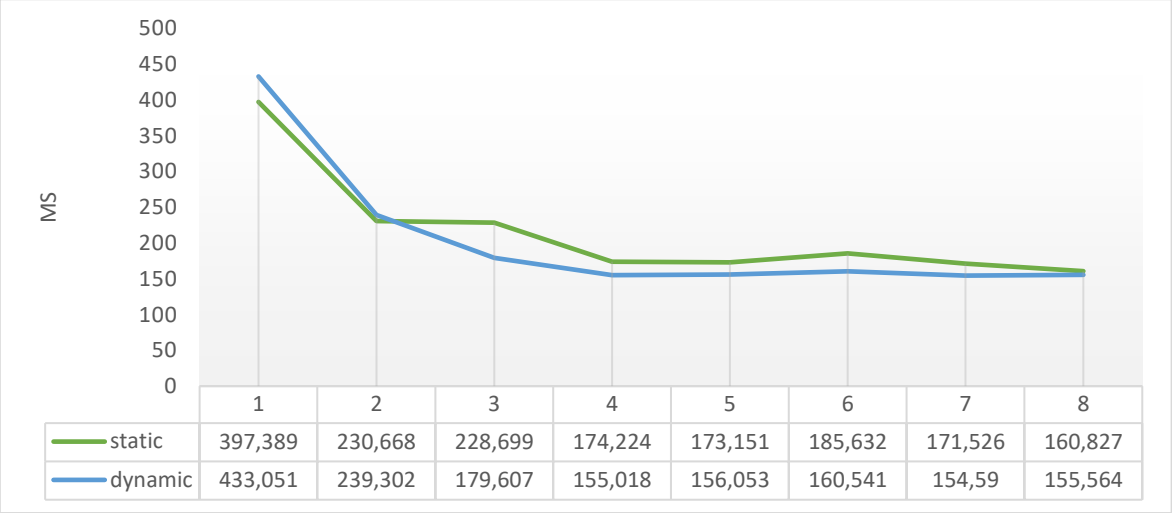
И соответственно все аргументы подавались в mainpnm.exe. Например, .\mainpnm.exe 8 .\BB.ppm rgb2.ppm 0.1

Также исходный файл должен располагаться в той же папке, что и программа.

Графики времени работы программы

Вычисление времени будет проводиться на моём ноутбуке с процессором AMD Ryzen 3 3200U 2.60 GHz с 2 ядрами и 4 потоками. Сравнение всех параметров происходило по обработке файла BB.ppm

(размером в 23,7 МБ) в rgb2.ppm с коэффициентом 0.1. Зависимость различных параметров schedule можно наблюдать на рисунке 7.



	1	2	3	4	5	6	7	8
static	397,389	230,668	228,699	174,224	173,151	185,632	171,526	160,827
dynamic	433,051	239,302	179,607	155,018	156,053	160,541	154,59	155,564

Рисунок 7 – Результаты работы программы с разными параметрами schedule

Видно по графику, что с увеличением потоков при одинаковом параметре schedule уменьшается время выполнения программы, что вполне логично. Но с каждым переходом выгода во времени становится всё меньше. Так же при одинаковом количестве потоков dynamic показывает себя лучше, чем static. Потенциально распараллеливание моей программы увеличивает скорость работы в 2-2,5 раза. Теперь сравним результаты с выключенным OpenMP и включённым 1 потоком на рисунке 8.

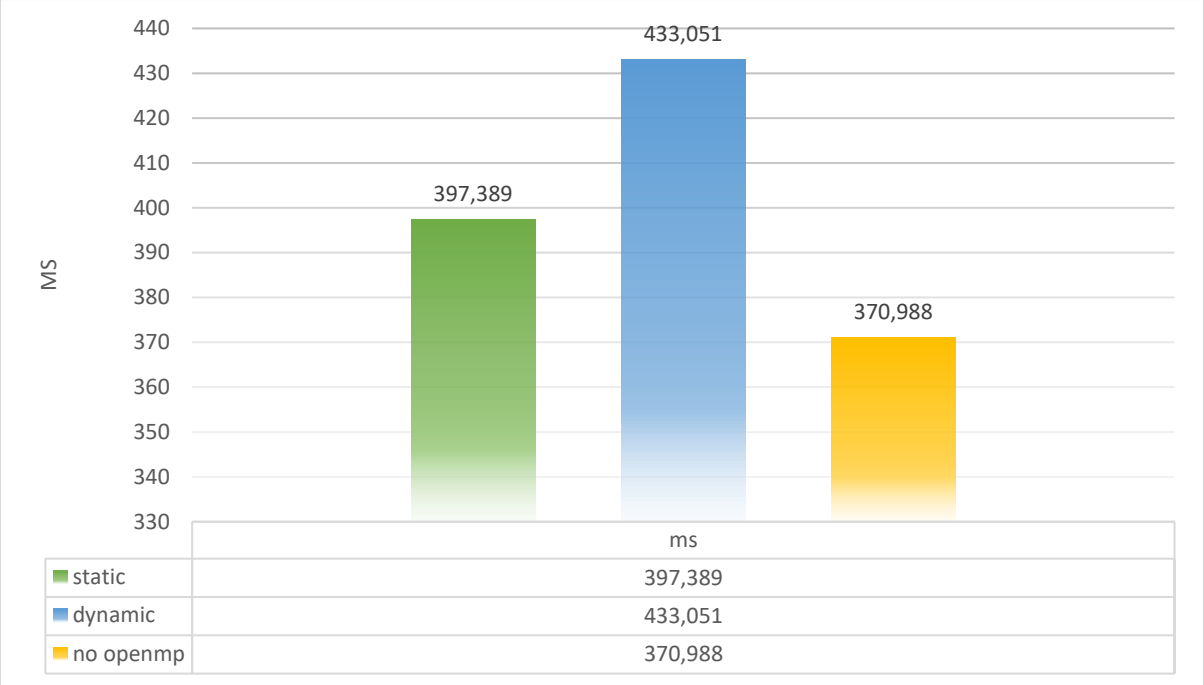


Рисунок 8 – Скорость работы с выключенным OpenMP и включённым 1 потоком.

В среднем, OpenMP с одним потоком требует немного больше времени нежели решение без OpenMP. Оно и понятно, помимо выполнения нашей задачи мы ещё тратим время на то, чтобы поддерживать один поток и из-за этого мы теряем во времени.

Листинг

g++ (GCC) 11.2.0

pnmParser.cpp

```
#include <iostream>
#include <omp.h>
#include <stdio.h>
#include <vector>
#include <fstream>

void getMinMax(float coef, const std::vector<unsigned char> &file, int
startOfBytes, int &amin, int &amax,
```

```
                unsigned char type, int s);

int getMax(const unsigned int *R, unsigned int ignor);

int getMin(unsigned int R[256], unsigned int ignor);

using namespace std;

std::vector<unsigned char> readFile(const std::string &filename) {
    std::streampos fileSize;
    std::ifstream file(filename, std::ios::binary);
    file.seekg(0, std::ios::end);
    fileSize = file.tellg();
    file.seekg(0, std::ios::beg);
    std::vector<unsigned char> fileData(fileSize);
    file.read((char *) &fileData[0], fileSize);
    return fileData;
}

int main(int argc, char **argv) {
    if (argc < 5) {
        fprintf(stderr, "You should type 4 arguments! Number of threads,
inputFile, outputFile and coefficient!");
        exit(0);
    }
    int num_of_threads = atoi(argv[1]);
    if (num_of_threads < 0) {
        fprintf(stderr, "Number of threads should be more or equal
zero!");
        exit(0);
    }
    omp_set_num_threads(atoi(argv[1]));
    std::string inputFile(argv[2]);
    std::string outputFile(argv[3]);
    float coef = std::stof(argv[4]);
    if (coef < 0 or coef >= 0.5) {
```

```
        fprintf(stderr, "Coefficient should be between 0 and 0.5!");
        exit(0);
    }

    //printf("%d %s %s %f\n", num_of_threads, inputFile.c_str(),
outputFile.c_str(), coef);
    vector<unsigned char> file = readFile(inputFile);
    double start_time = omp_get_wtime();
    if (file.size() == 0) {
        fprintf(stderr, "File is empty!");
        exit(0);
    }
    if (file[0] != 'P') {
        fprintf(stderr, "File's magic numbers should start with 'P'");
        exit(0);
    }
    if (file[1] != '5' && file[1] != '6') {
        fprintf(stderr, "File's magic numbers should end with '5' or
'6'");
        exit(0);
    }
    unsigned char type = file[1];
    if (file[2] != 0x0A) {
        fprintf(stderr, "Line separator is missing");
    }
    int index = 3;
    string width;
    while (file[index] != 0x20) {
        width += file[index++];
    }
    index++;
    string height;
    while (file[index] != 0x0A) {
        height += file[index++];
    }
    index++;
```

```
    string pixels;
    //printf("width: %s and height: %s. And of course type: %c\n",
width.c_str(), height.c_str(), type);
    while (file[index] != 0x0A) {
        pixels += file[index++];
    }
    if (pixels != "255") {
        fprintf(stderr, "This program only works with 255 pixels rad.
actual: %s", pixels.c_str());
        exit(0);
    }
    index++;
    int startOfBytes = index;
    int amin;
    int amax;
    getMinMax(coef, file, startOfBytes, amin, amax, type, num_of_threads);
    if (amin == amax) {
        fprintf(stderr, "You have overdone contrast! Please make coef
less!");
        exit(0);
    }
    //printf("amin : %d, amax: %d\n", amin, amax);
    float cc = 255.0 * 1 / (amax - amin);
#pragma omp parallel
    {
#pragma omp for schedule(dynamic, 512)
//#pragma omp for schedule(static, 512)
        for (int i = startOfBytes; i < file.size(); i++) {
            int answer = (file[i] - amin) * cc;
            if (answer < 0) {
                file[i] = 0;
            } else if (answer > 255) {
                file[i] = 255;
            } else {
                file[i] = answer;
            }
```

```
        }
    }
    printf("Time (%i thread(s)): %g ms\n", num_of_threads,
(omp_get_wtime() - start_time) * 1000);
    ofstream fout(outputFile, std::ios::binary);
    for (unsigned char i: file) {
        fout << i;
    }
}


void getMinMax(float coef, const vector<unsigned char> &file, int
startOfBytes, int &amin, int &amax, unsigned char type,
          int num_of_threads) {
    if (type == '5') {
        unsigned int bytes[256] = {};
        int numOfBytes = file.size() - startOfBytes;
#pragma omp parallel
        {
#pragma omp for schedule(dynamic, 512) reduction(+:bytes)
//#pragma omp for schedule(static, 512) reduction(+:bytes)
            for (int i = startOfBytes; i < file.size(); i++) {
//#pragma omp atomic
                bytes[file[i]] += 1;
            }
        }
        //printf("Number of total bytes: %d\nAnd fucking bytes: \n",
numOfBytes);
        unsigned int ignor = coef * numOfBytes;
        //printf("We should ignore %d bytes\n", ignor);
        amin = getMin(bytes, ignor);
        amax = getMax(bytes, ignor);
    } else if (type == '6') {
        unsigned int R[256] = {};
        unsigned int G[256] = {};
        unsigned int B[256] = {};
```

```
        int numOfBytes = file.size() - startOfBytes;
#pragma omp parallel
        {
#pragma omp for schedule(dynamic, 512) reduction(+:R, G, B)
//#pragma omp for schedule(static, 512) reduction(+:R, G, B)
            for (int i = startOfBytes; i < file.size(); i += 3) {
//#pragma omp atomic
                R[file[i]] += 1;
//#pragma omp atomic
                G[file[i + 1]] += 1;
//#pragma omp atomic
                B[file[i + 2]] += 1;
            }
        }

        //printf("Number of total bytes: %d\nAnd fucking bytes: \n",
numOfBytes);
        //        for (int i = 0; i < 256; i++) {
        //            printf("%x : %d\n", i, R[i]);
        //        }
        unsigned int ignor = coef * (numOfBytes / 3);
        //printf("We should ignore %d bytes\n", ignor);
        int Rmin = getMin(R, ignor);
        int Gmin = getMin(G, ignor);
        int Bmin = getMin(B, ignor);
        int Rmax = getMax(R, ignor);
        int Gmax = getMax(G, ignor);
        int Bmax = getMax(B, ignor);
        amin = min(Rmin, min(Gmin, Bmin));
        amax = max(Rmax, max(Gmax, Bmax));
    } else {
        fprintf(stderr, "Unknown type: %c", type);
        exit(0);
    }
}
```

```
int getMin(unsigned int R[256], unsigned int ignor) {
    int min;
    int sum = 0;
    for (int i = 0; i < 256; i++) {
        sum += R[i];
        if (sum > ignor) {
            min = i;
            break;
        }
    }
    return min;
}

int getMax(const unsigned int *R, unsigned int ignor) {
    int max;
    int sum = 0;
    for (int i = 255; i >= 0; i--) {
        sum += R[i];
        if (sum > ignor) {
            max = i;
            break;
        }
    }
    return max;
}
```