САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКАЙ

УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

По домашней роботе №5

«OpenMP»

Выполнил: Зайнидинов Мирзофирдавс Шавкатович

Студент группы M313D

Санкт-Петербург

2020

Цель работы: <Знакомство со стандартом распараллеливания команд OpenMP>

Инструментарий и требования к работе: <C++, компилятор Visual Studio 2019>

Теоретическая часть

OpenMP – открытый стандарт для распараллеливания программ. Стандарт OpenMP был разработан в 1997 году как **API** (application programming interface – программный интерфейс приложения) ориентированный на написание портируемых многопоточных приложений. Сначала он был разработан для языка **Fortran**, но позднее включил в себя **C** и **C++**. В 2014 году вышла версия **OpenMP** 4.0. Код за годом оно развивается, и готова представит всё новые версии. **OpenMP** реализует параллельные вычисления с помощью многопоточности, в которых главный поток создает набор подчиненных потоков. Главный поток – **Master**, подчиненный поток – **Slave**. Во время многопоточности задачу который выполняет алгоритм разделяется между этими потоками. Потоки выполняются параллельно на машинах с несколькими процессорами, количество процессоров не обязательно должна быть больше или равно количеству потоков. Задачи выполняемые потоками так же как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка программирования. Препроцессор – это компьютерная программа, принимающее на вход данные и выдающая данные, для входа, для другой программы. Количество создаваемых потоков может регулироваться так и самой программой при помощи базовых библиотек, процедур, которые используются, так и извне при помощи переменных окружения. **OpenMP** прост в использовании и включает лишь два базовых типа конструкций: директивы **pragma** и функции исполняющей среды **OpenMP**. Директивы **pragma** указывают компилятору реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с **#pragma omp**. Функции

OpenMP служат в основном для изменения и получения параметров среды. Кроме того, **OpenMP** включает **API** – функции для поддержки некоторых типов синхронизации. Чтобы задействовать эти функции библиотеки **OpenMP** периода выполнения, в программу нужно включить заголовочный файл **omp.h**. Хотя директив **OpenMP** много, самая важная и распространенная директива — **parallel**. Она создает параллельный регион для следующего за ней структурированного блока. Эта директива сообщает компилятору, что структурированный блок кода должен быть выполнен параллельно, в нескольких потоках. Каждый поток будет выполнять один и тот же поток команд, но не один и тот же набор команд — все зависит от операторов, управляющих логикой программы, таких как **if-else**. В качестве примера смотрите на рисунок №1.

```
#pragma omp parallel
{
    printf("Hello World\n");
}
```

Рисунок №1

В двухпроцессорной системе можно рассчитывать получить ответ:

Hello World

Hello World

Тем не менее, результатом может быть:

HellHell oo WorWlodrl

Это из – за того, что два выполняемых параллельно потока могут попытаться вывести файл одновременно. Когда два или более потоков пытаются проводить разные операции с одним файлом возникают такого рода ошибки. Разрабатывая параллельные программы, надо уметь понимать, какие данные являются общими (shared), а какие частными (private) – от это зависит не

только производительность, но и корректная работа программы. В **OpenMP** различие очевидны и их можно настроить вручную. Общие переменные доступны всем потокам из группы, поэтому их изменения в одном потоке влияют на другие потоки. А для частных переменных, каждый поток располагает их отдельным экземпляром и их изменения в различных потоках, не влияет на других. По умолчанию все переменные в параллельных регионах общие, но есть исключения:

1. Частными являются индексы параллельных цикло **for**
2. Частными являются локальные переменные блоков параллельных регионов
3. Частными будут любые переменные, указанные в разделах **firstprivate**, **lastprivate**, **reduction**.

Как правило **OpenMP** используется для распараллеливания циклов, но **OpenMP** поддерживает параллелизм и на уровне функций. Этот механизм называется секциями **OpenMP**. Он довольно прост и часто бывает полезен. При одновременном выполнении нескольких потоков часто возникает необходимость их синхронизации. **OpenMP** поддерживает несколько типов синхронизации, помогающих в разных ситуациях.

1. Неявная барьерная синхронизация, которая выполняется в конце каждого параллельного региона для всех сопоставленных с ним потоков. Механизм барьерной синхронизации таков, что пока все потоки не достигнут конца параллельного региона, ни один поток не сможет перейти его границу.
2. Явная барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявно синхронизацией. Для этого надо включить в код директиву **#pragma omp barrier**. В параллельных регионах часто встречаются блоки кода, доступ к которым желательно предоставлять только одному потоку, — например, блоки кода, отвечающие за запись данных в файл. Во многих таких ситуациях

не имеет значения, какой поток выполнит код, важно лишь, чтобы этот поток был единственным. Для этого в **OpenMP** служит директива **#pragma omp single**.

Помимо уже описанных директив **OpenMP** поддерживает ряд полезных подпрограмм. Они делятся на три обширных категории:

1. Функции исполняющей среды блокировки
2. Функции исполняющей среды Синхронизации
3. Функции исполняющей среды работы с таймерами

По умолчанию в **OpenMP** для планирования параллельного выполнения циклов **for** применяется алгоритм, называемый статическим планированием (**static scheduling**). Это означает, что все потоки из группы выполняют одинаковое число итераций цикла. Если **n** — число итераций цикла, а **T** — число потоков в группе, каждый поток выполнит **n/T** итераций. Однако **OpenMP** поддерживает и другие механизмы планирования, оптимальные в разных ситуациях: динамическое планирование (**dynamic scheduling**), планирование в период выполнения (**runtime scheduling**) и управляемое планирование (**guided scheduling**). Чтобы задать один из этих механизмов планирования, используется раздел **schedule** в директиве **#pragma omp for** или **#pragma omp parallel for**. Смотрите на рисунок №2

```
schedule(алгоритм планирования[, число итераций])
```

Рисунок №2

При динамическом планировании каждый поток выполняет указанное число итераций. Если это число не задано, по умолчанию оно равно 1. После того как поток завершит выполнение заданных итераций, он переходит к следующему набору итераций. Так продолжается, пока не будут пройдены все итерации. Последний набор итераций может быть меньше, чем изначально заданный.

Если указать директиву **#pragma omp for schedule(dynamic, 15)**, цикл **for** из 100 итераций может быть выполнен четырьмя потоками (смотрите на рисунок №3)

```
Поток 0 получает право на выполнение итераций 1-15
Поток 1 получает право на выполнение итераций 16-30
Поток 2 получает право на выполнение итераций 31-45
Поток 3 получает право на выполнение итераций 46-60
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 61-75
Поток 3 завершает выполнение итераций
Поток 3 получает право на выполнение итераций 76-90
Поток 0 завершает выполнение итераций
Поток 0 получает право на выполнение итераций 91-100
```

Рисунок №3

А если указать директиву **#pragma omp for schedule(quided, 15)**, цикл **for** из 100 итераций может быть выполнен четырьмя потоками (смотрите на рисунок №4)

```
Поток 0 получает право на выполнение итераций 1-25
Поток 1 получает право на выполнение итераций 26-44
Поток 2 получает право на выполнение итераций 45-59
Поток 3 получает право на выполнение итераций 60-64
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 65-79
Поток 3 завершает выполнение итераций
Поток 3 получает право на выполнение итераций 80-94
Поток 2 завершает выполнение итераций
Поток 2 получает право на выполнение итераций 95-100
```

Рисунок №4

Динамическое и управляемое планирование хорошо подходят, если при каждой итерации выполняются разные объемы работы или если одни процессоры более производительны, чем другие. При статическом

планировании нет никакого способа, позволяющего сбалансировать нагрузку на разные потоки. При динамическом и управляемом планировании нагрузка распределяется автоматически — такова сама суть этих подходов. Как правило, при управляемом планировании код выполняется быстрее, чем при динамическом, вследствие меньших издержек. Хотя кажется что **OpenMP** не очень то и сложно использовать. У него есть свои тонкости, где нужно уделять повышенное внимание. Приведем некоторые примеры тонкостей:

1. индексная переменная самого внешнего параллельного цикла **for** является частной, а индексные переменные вложенных циклов **for** по умолчанию общие. При работе с вложенными циклами обычно требуется, чтобы индексы внутренних циклов были частными. Используйте для этого раздел **private**.
2. Если работаем с **OpenMP** на **C++**, стоит быть осторожным при генераций исключений. Если надо исключение генерируется в параллельном регионе, она должна быть обработана в том же регионе.

Практическая часть

Для практической работы я выбрал вариант 7, распараллеливания алгоритма нахождения определителя квадратной матрицы. Вычисляю определитель квадратной матрицы приводя ее к треугольному виду, после этого перемножая диагональ квадратной матрицы для нахождения ее определителя. Данный алгоритм работает за временную оценку $\mathbf{O(N^3)}$ где **N** – это размер квадратной матрицы. Ну алгоритм таков что, надо берем произвольную строку, можно делать две операции:

1. Можно умножать элементы этой строки на какое ни будь число.
2. Сложить или вычесть строки.

Делаем произвольное число таких операций пока наша матрица не будет вида треугольника (смотрите на рисунок №5).

a1	a2	a3	a4	a5	a6	a7
o	a8	a9	a10	a11	a12	a13
o	o	a14	a15	a16	a17	a18
o	o	o	a19	a20	a21	a22
o	o	o	o	a23	a24	a25
o	o	o	o	o	a26	a27
o	o	o	o	o	o	a28

Рисунок №5

Будем рассматривать значения от 1 потока до 4. Запишем результаты в таблицу. В каждой ячейке делаем три вычисления времени работы программы (смотрите на рисунок №6). Будем все это вычислять для случайно сгенерированный квадратной матрицы размера 1500.

	1 поток	2 поток	3 поток	4 поток
static	4.444 4.445 4.424	2.124 2.009 2.406	2.961 3.001 2.963	2.125 2.300 2.285
dynamic	4.521 4.586 4.623	2.521 2.513 2.590	3.125 3.159 3.253	2.400 2.425 2.430
guided	4.700 4.715 4.722	2.615 2.525 2.611	3.005 3.075 3.126	2.455 2.470 2.399
parallel with OpenMP	4.275 4.266 4.251			

Рисунок №6

Чтобы компилировать файл на командной строке:

mingw32-g++.exe -static -fopenmp -c hw5.cpp -o hw5.o

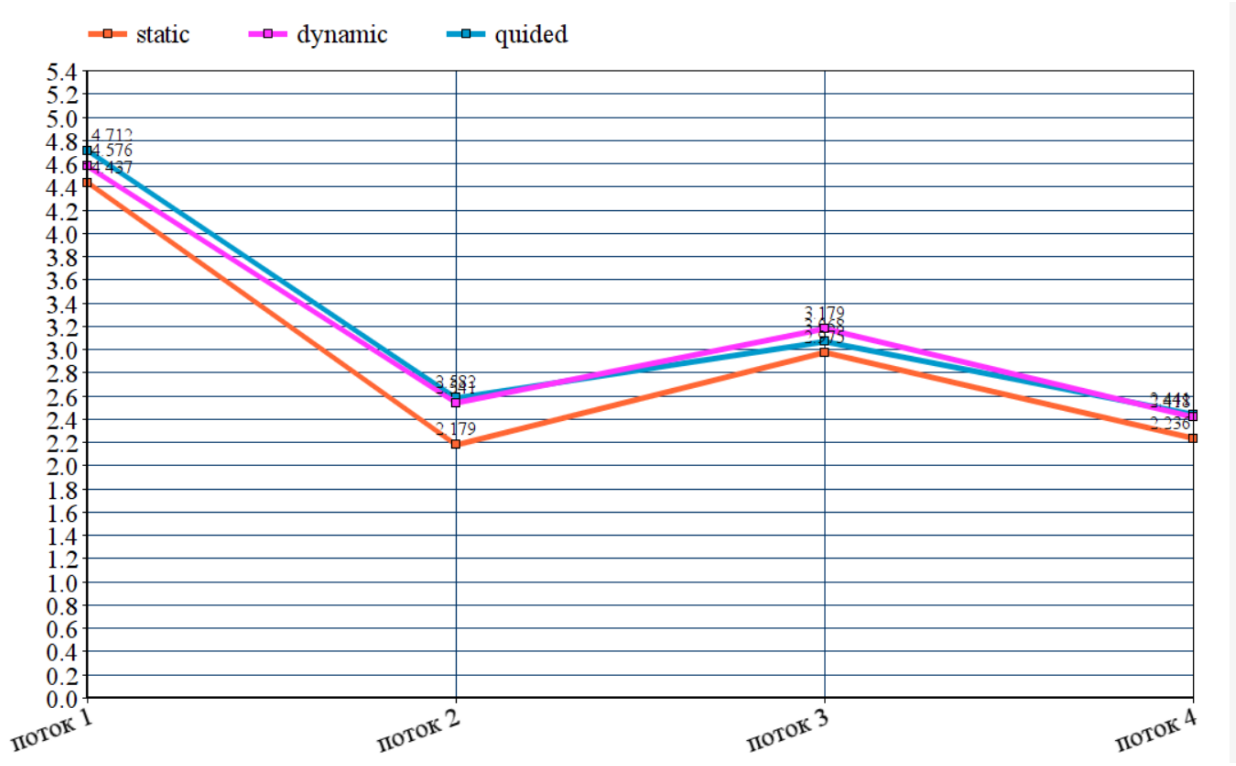
mingw32-g++.exe -static -fopenmp -o hw5.exe hw5.o

Компилирования без **OpenMP:**

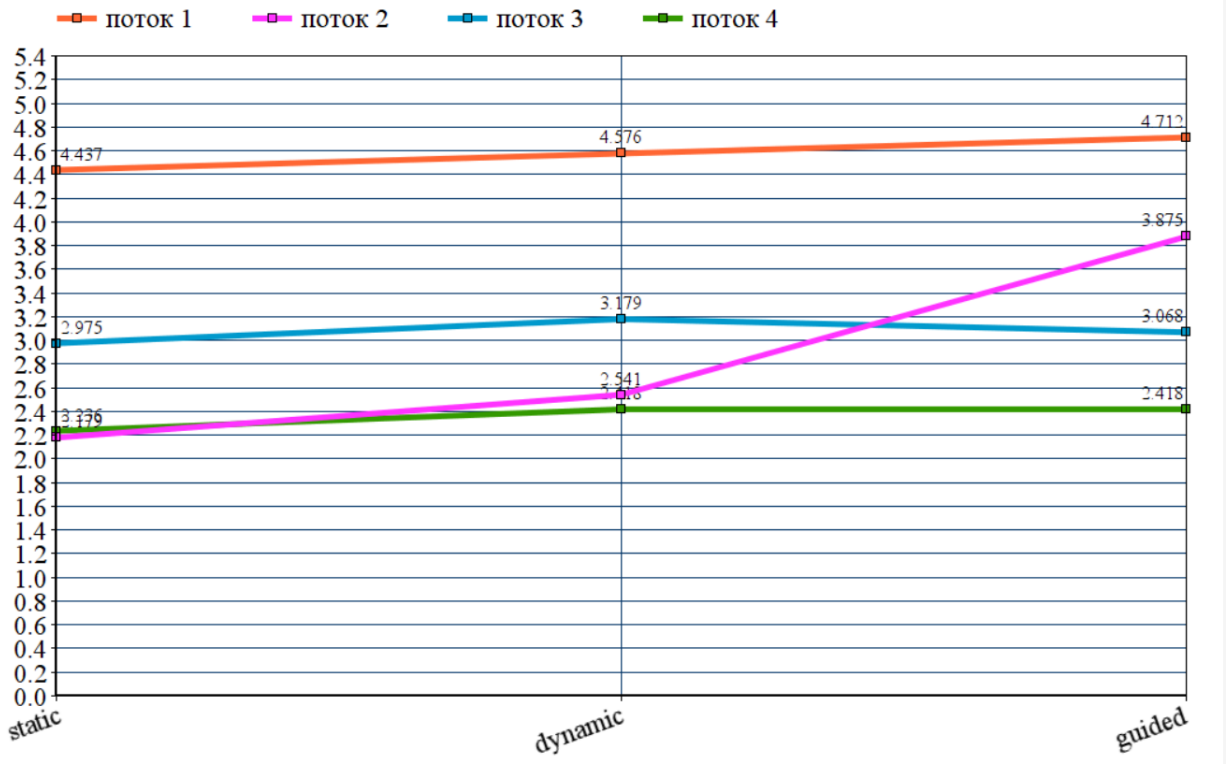
mingw32-g++.exe -static -c hw5.cpp -o hw5.o

mingw32-g++.exe -static -o hw5.exe hw5.o

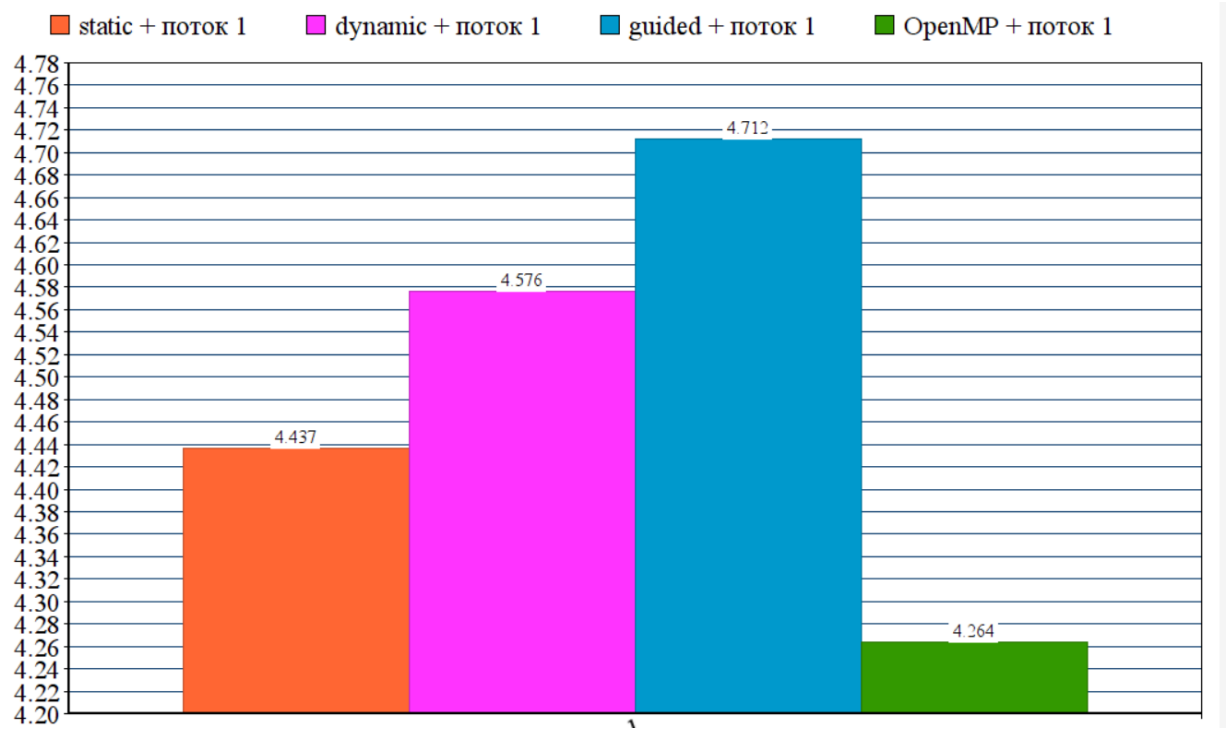
Берем среднее арифметическое из значений и пишем результаты в таблицы которые нам надо сделать по условию отчета.



При различных значениях числа потоков при одинаковом параметре **schedule**



При одинаковом числе потоков различных параметров **schedule**



С выключенным **OpenMP** с 1 потоком и с включенным **OpenMP** с 1 потоком

Код:

Язык: C++

Компилятор: mingw 32 bit version 6.3.0

hw5.cpp

```
#include <iostream>

#include <omp.h>

#include <sstream>

#include <cstdio>

#include <chrono>


using namespace std;


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

    if (argc <= 2) {

        printf("Usage: <program name> <number of threads> <input file> [<output file>].\n");

        return 0;

    }


    stringstream convert(argv[1]);

    int threads;

    if (!(convert >> threads)) {

        printf("Error: can not input number of threads.\n");

        return 0;

    }


    int maxThreads = omp_get_max_threads();

    if (threads < 0 or maxThreads < threads) {
```

```
        printf("Incorrect value number of threads: should be from 0 to %i.\n",
maxThreads);
        return 0;
    }
    if (threads && threads < maxThreads) {
        omp_set_num_threads(threads);
    } else {
        threads = maxThreads;
    }

    FILE *file;
    file = fopen(argv[2], "r");
    if (file == NULL) {
        printf("Can not open input file.\n");
        return 0;
    }

    fclose(file);
    freopen(argv[2], "r", stdin);

    bool writing = false;
    if (argc == 4) {
        file = fopen(argv[3], "w");
        if (file == NULL) {
            writing = false
            printf("Can not open output file.\n");
            return 0;
        }
        writing = true;
```

```
}

int n;

double det;

cin >> n;
auto** ma = new double*[n];
for (int i = 0; i < n; ++i) {
    ma[i] = new double[n];
    for (int j = 0; j < n; ++j) {
        cin >> ma[i][j];
    }
}

auto start = std::chrono::steady_clock::now();

det = 1;

for (int i = 0; i < n; ++i) {
    if (ma[i][i] == 0) {
        int f = i;
        for (int j = i + 1; j < n; ++j) {
            if (ma[j][i] != 0) {
                f = j;
                break;
            }
        }
        if (f == i) {
```

```
                det = 0;
                break;
            } else {
                det = -det;
                swap(ma[i], ma[f]);
            }
        }
        det *= ma[i][i];
        #pragma omp parallel for schedule(dynamic)
        for (int j = i + 1; j < n; ++j) {
            if (ma[j][i] == 0) continue;
            double zn = -ma[j][i] / ma[i][i];
            ma[j][i] = 0;
            for (int k = i + 1; k < n; ++k) {
                ma[j][k] += ma[i][k] * zn;
            }
        }
        delete [] ma[i];
    }

    auto finish = std::chrono::steady_clock::now();
    auto elapsed_ms = std::chrono::duration_cast<std::chrono::milliseconds>(finish - start);

    delete [] ma;
    if (writing) {
        fprintf(file, "Determinant: %g\n", det);
        fclose(file);
    }
```

```
    printf("Determinant: %g\n", det);

    printf("\nTime (%i thread(s)): %f ms\n", threads, (double)elapsed_ms.count());

    return 0;

}
```