Лабораторная работа №5	ФИО: Кинзябулатов Эдуард Шамилевич
Название работы: OpemMp	Группа: M3137

Цель работы: знакомство с основами многопоточного программирования.

Ссылка на репозиторий: https://github.com/skkv-itmo/itmo-comp-arch-2023-omp-Tortik3000

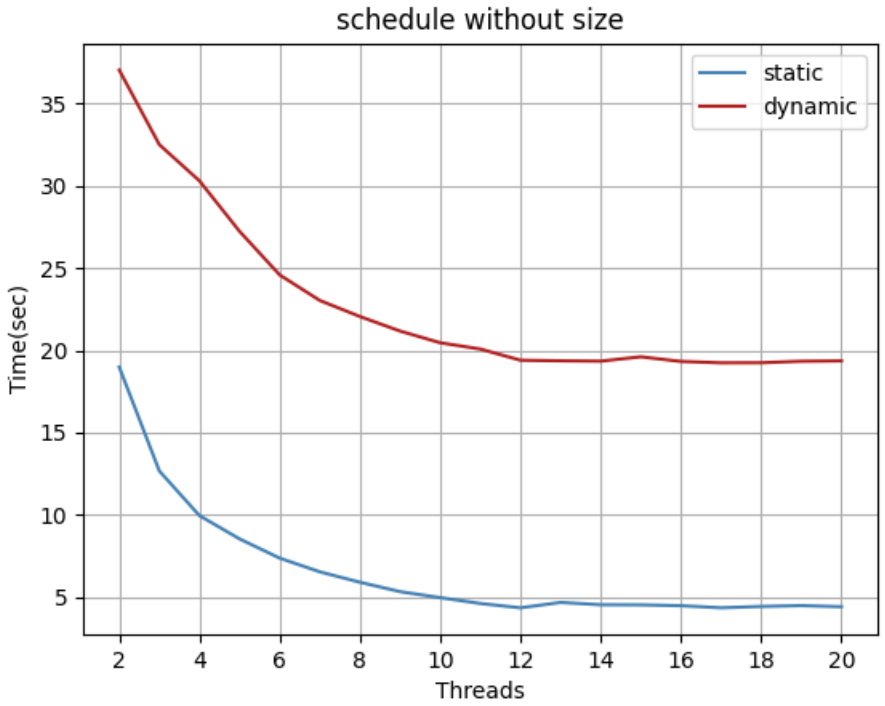
Инструментарий и требования к работе: C++20

Результат работы:

```
Time (12 thread(s)): 4457.94 ms
36.0001 36.0003

Процессор AMD Ryzen 5 5500U with Radeon Graphics
```

Усреднение значений бралось по 4 запускам.

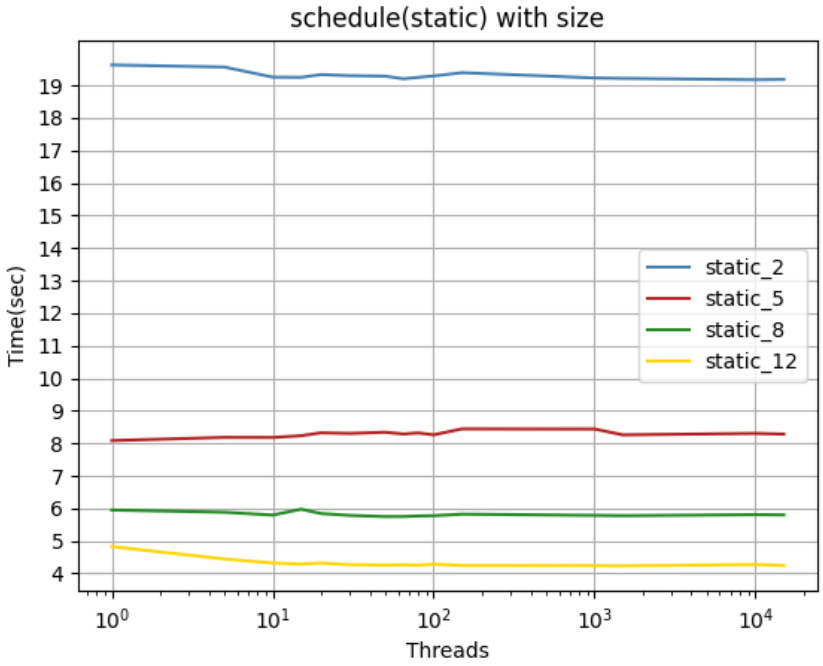
Время работы при различных значениях числа потоков при одинаковом параметре schedule:

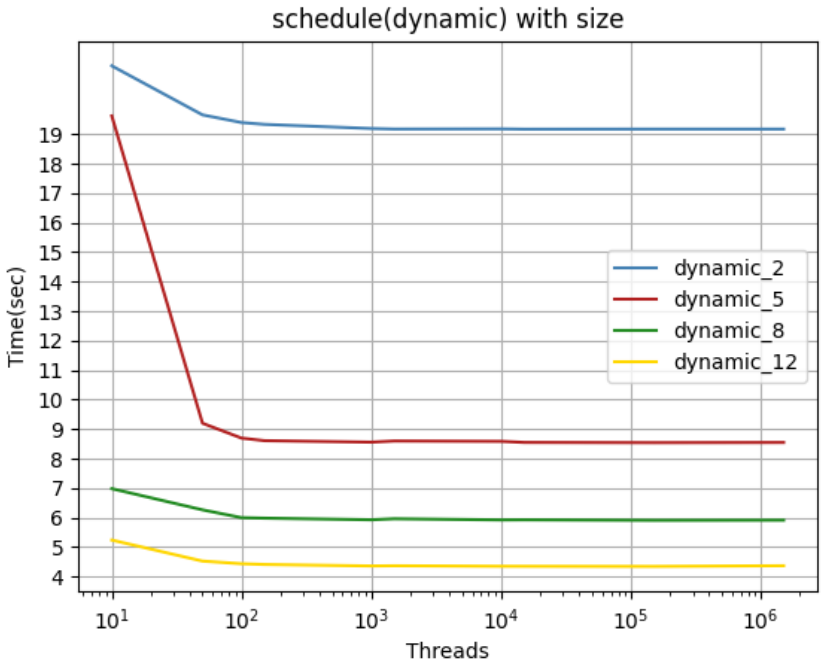


Из графика видно что программа работает быстрее всего с 12 потоками при использовании static.

Также видно что static работает быстрее dynamic, это происходит из-за того, что при использовании dynamic каждый поток получает определенное количество итераций для выполнения равное chunk_size. После завершения всех итераций, поток запрашивает новый блок итераций и на это распределение итераций также тратится время. Поэтому при маленьких значениях chunk_size время сэкономленное на том, что потоки не простаивают, незначительное по сравнению с временем затрачиваемым на распределение итераций.

Время работы при одинаковых значениях числа потоков при различных параметрах schedule(static/dynamic):





Из графиков видно, что лучшее время получается при использовании static с Schunk_size = 10^3

Также static дает небольшой выигрыш по времени из-за того, что все итерации выполняются примерно за одинаковое время, поэтому динамическое распределение итераций не дает улучшения времени, но время тратится на распределение итераций.

Время работы с одним потоком: 38328.9 ms

Время работы с выключенным OpenMp: 37992.36 ms

Здесь видно, что программа с включенным одним потоком работает медленнее,чем код с выключенным OpenMp. Это происходит из-за того, что тратится время на создание и закрытие потока.

Описание инструкций OpenMP для распараллеливания команд.

(приложение№1 2.3)

1. #pragma omp parallel [clause[[,]clause] ...]{}

   Определяет блок программы который будет выполняться несколькими потоками.

   Может работать с аргументом if(scalar-expression), программа будет выполняться параллельно если If отсутствует или принимает значение true.

(приложение№1 2.4)

2. #programa omp for [clause[[,]clause] ...]{}

   Указывает на то, что следующий цикл будет выполнятся несколькими потоками.

Может работать с аргументом schedule(*kind* ,*chunk_size*). В лабораторной kind принимает значения static/dynamic. Если kindпринимает значение static, итерации делятся на блоки размера равного chunck_size. Потом эти блоки равномерно распределяются между потоками до начала выполнения цикла. Если chunk_size не указан, компилятор будет делить итерации цикла равномерно между потоками.

Если kind принимает значение dynamic итерации цикла динамически распределяются между потоками во время выполнения программы. Когда один поток завершает обработку определенной итерации, он запрашивает следующий доступный блок итераций размера chunk_size для обработки. Если chunk_size *не задано*, значение по умолчанию равно 1. dynamic полезен когда объем работы в итерациях цикла неравномерный.

(приложение№1 2.6.2)

3. #program omp critical

   Определяет конструкцию, которая будет выполняться одновременно не более чем одним потоком. Это важно, когда нескольким потокам требуется доступ к общим ресурсам, и это обеспечивая целостность данных.

Описание работы кода:

Считывание координат точек и количество точек для метода Монте-Карло

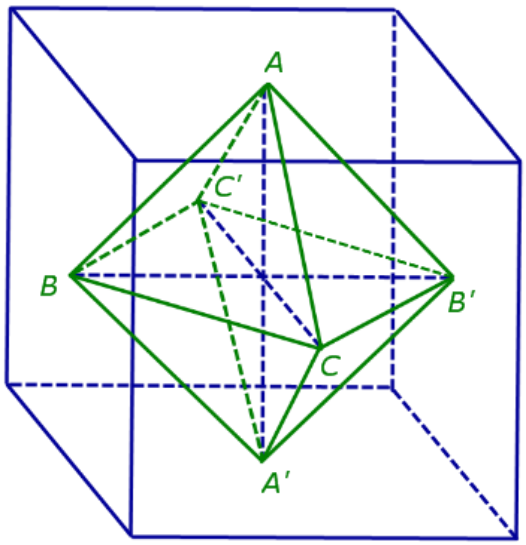
```
int n;
int streamCount = std::atoi( nptr: argv[1]);
double point0[3];
double point1[3];
double point2[3];

try {
    FILE *fin = fopen( name: argv[2],  type: "rb");
    fscanf(fin, "%i", &n);
    fseek(fin, 3, SEEK_CUR);

    for (int i = 0; i < 3; i++) {
        double coord;
        if (i == 0) {
            for (int j = 0; j < 3; j++) {
                fscanf(fin, "%lf", &coord);
                point0[j] = coord;
            }
        } else if (i == 1) {
            for (int j = 0; j < 3; j++) {
                fscanf(fin, "%lf", &coord);
                point1[j] = coord;
            }
        } else {
            for (int j = 0; j < 3; j++) {
                fscanf(fin, "%lf", &coord);
                point2[j] = coord;
```

```
            }
        } else {
            for (int j = 0; j < 3; j++) {
                fscanf(fin, "%lf", &coord);
                point2[j] = coord;
            }
        }
        fseek(fin, 4, SEEK_CUR);
    }
    fclose(fin);
}catch (std::exception exception){
    std::cout<< "File format exception";
}catch (const std::ios_base::failure exception) {
    std::cout<< "File not faund";
}
```

Далее идет расчет координат ребер и их длин

```
double tstart = omp_get_wtime();

double edge0[3], edge1[3], edge2[3];
for (int i = 0; i < 3; i++) {
    edge0[i] = (point1[i] - point0[i]);
    edge1[i] = (point1[i] - point2[i]);
    edge2[i] = (point0[i] - point2[i]);
}

double moduleEdge0, moduleEdge1, moduleEdge2;
moduleEdge0 = sqrt(edge0[0] * edge0[0] + edge0[1] * edge0[1] + edge0[2] * edge0[2]);
moduleEdge1 = sqrt(edge1[0] * edge1[0] + edge1[1] * edge1[1] + edge1[2] * edge1[2]);
moduleEdge2 = sqrt(edge2[0] * edge2[0] + edge2[1] * edge2[1] + edge2[2] * edge2[2]);


double min0 = moduleEdge0;
if (min0 > moduleEdge1) {
    min0 = moduleEdge1;
}

min0 = min0 / sqrt(2);
if (moduleEdge0 == moduleEdge1 && moduleEdge0 == moduleEdge2) {
    min0 = moduleEdge0 / sqrt(2);
}
double v = min0 * min0 * min0 * 8;
```

min0 = BB’/2, v – объем параллелепипеда в котором будем случайно генерировать точки.



Потом начинается блок который будет выполняться параллельно если количество потоков не равно 0

Внутри блока создается переменная localM для каждого потока, так как если сразу прибавлять значения к исходному счетчику m некоторые значения не будут учитываться.

Далее в блоке omp for параллельно выполняются итерации цикла,

на каждой итерации генерируется координата точки с помощью mt19937

и проверяется, что точка лежит между 8 плоскостями задающими октаэдр.

Потом переменные localM каждого потока прибавляются к исходному счетчику m,

В блоке critical, где одновременно может быть только один поток.

```
    std::random_device rd;
#pragma omp parallel\
    if(streamCount != -1)
        {
            std::mt19937 gen( sd: rd());
            int localM = 0;
            double x, y, z;
            if (streamCount == 0) {
                streamCount = omp_get_num_threads();
            }
#pragma omp for schedule(static, 1000)
        for (int i = 0; i < n; i++) {
            x = gen() / 1.0 / UINT32_MAX * 2 * min0 - min0;
            y = gen() / 1.0 / UINT32_MAX * 2 * min0 - min0;
            z = gen() / 1.0 / UINT32_MAX * 2 * min0 - min0;
            if (x < 0) {x = -x;}
            if (y < 0) {y = -y;}
            if (z < 0) {z = -z;}
            if (x + y + z <= min0) {
                localM++;
            }
        }
#pragma omp critical
            {
                m += localM;
            }
        }
```

Потом идет подсчет ответа: (всех точеки)/(точки попавшие в октаэдр) *(объем параллепипеда)

И вывод ответа

```
double ans = v * m / n;
double ansAnalytic = v / 6;

double tend = omp_get_wtime();
if (streamCount == -1) {
    streamCount += 1;
}

FILE *fout = fopen( name: argv[3],  type: "wb");
fprintf(fout, "%g %g\n", ansAnalytic, ans);
fclose(fout);
printf("Time (%i thread(s)): %g ms\n", streamCount, (tend - tstart) * 1000);
```

Функции генерации случайных координат(приложение 2):

```
void randomCoord(int n, double range, double* coords){
    double f3 = 1.22074408460575947536;
    double a1 = 1.0 / f3;
    double a2 = 1.0/(f3 * f3);
    double a3 = 1.0/(f3 * f3 * f3);

    coords[0] = ((random0 + a1 * n) - int(random0 + a1 * n)) * 2 * range - range ;
    coords[1] = ((random0 + a2 * n) - int(random0 + a2 * n)) * 2 * range - range ;
    coords[2] = ((random0 + a3 * n) - int(random0 + a3 * n)) * 2 * range - range ;

}
```

Используемые источники:

1.https://learn.microsoft.com/ru-ru/cpp/parallel/openmp/2-directives?view=msvc-170#24-work-sharing-constructs (конструкции OpenMp)