МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ» (ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики **Кафедра** математических методов исследования операций

ОТЧЁТ

по дисциплине

«**Основы параллельного программирования**»

Лабораторная работа № 3

Выполнил студент:
Задорожний Илья Владимирович
Курс 3 Группа 7 (ММИО)

Воронеж, 2024

1. Постановка задачи

Цель работы: Ознакомиться с директивами синхронизации.

Задача (10 вариант): Параллельное преобразование координат: Создайте программу, в которой каждый поток преобразует координаты точек 3D-объекта по заданной формуле (например, добавляя смещение). Используйте #pragma omp barrier для синхронизации после завершения преобразования.

2. Ход работы

1. **Буду хранить координату точки в структуре. Потом напишу функцию, которая реализует смещение 3д объекта (без использования параллельного программирования).**

```cpp
struct Point {
    double x, y, z;
};

void transform_points(std::vector<Point>& points, double offset_x, double offset_y, double offset_z) {
    for (auto& point : points) {
        point.x += offset_x;
        point.y += offset_y;
        point.z += offset_z;
    }
}
```

```cpp
int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    const int num_points = 100; // Количество точек
    std::vector<Point> points(num_points); // Создание вектора точек


    for (int i = 0; i < num_points; ++i) {
        points[i] = { double(i), double(i), double(i) }; // Каждая точка инициализируется значениями i
    }

    double offset_x = 1.0, offset_y = 2.0, offset_z = 3.0;

    double start_time = omp_get_wtime(); // Засекаем начальное время

    transform_points(points, offset_x, offset_y, offset_z);

    double end_time = omp_get_wtime(); // Засекаем конечное время

    double elapsed_time = double(end_time - start_time);// Время выполнения в секундах

    std::cout << "Sequential Time: " << std::fixed << std::setprecision(6) << elapsed_time << " seconds\n";

    return 0;
}
```

2. **Теперь решим задачу при помощи параллельного программирования.**

```cpp
#include <iostream>
#include <vector>
#include <omp.h>
#define NUM_THREADS 8

struct Point {
    double x, y, z;
};


void transform_points_parallel(std::vector<Point>& points, double offset_x, double offset_y, double offset_z) {
    int num_points = points.size();
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for
    for (int i = 0; i < num_points; ++i) {
        points[i].x += offset_x;
        points[i].y += offset_y;
        points[i].z += offset_z;
    }
#pragma omp barrier
}
```

```cpp
int main() {
    const int num_points = 1000000;
    std::vector<Point> points(num_points);

    for (int i = 0; i < num_points; ++i) {
        points[i] = { double(i), double(i), double(i) }; // Каждая точка инициализируется значениями i
    }


    double offset_x = 1.0, offset_y = 2.0, offset_z = 3.0;


    double start_time = omp_get_wtime(); // Засекаем начальное время

    transform_points_parallel(points, offset_x, offset_y, offset_z);

    double end_time = omp_get_wtime(); // Засекаем конечное время


    double elapsed_time = end_time - start_time; // Время выполнения в секундах

    std::cout << "Parallel Time: " << elapsed_time << " seconds\n";

    return 0;
}
```

Что происходит при использовании OpenMP:

1. Создание потоков: При входе в блок #pragma omp parallel for, OpenMP создает несколько потоков, каждый из которых выполнит часть работы.
2. Разделение работы: Цикл, который обрабатывает массив точек (в нашем случае это цикл по индексу i), автоматически делится между потоками. OpenMP распределяет итерации цикла по потокам таким образом, чтобы они выполнялись параллельно.
   Например, если у нас есть 4 потока и 1000 точек, то каждому потоку может быть передана работа с 250 точками. Поток 0 обработает первые

250 точек, поток 1 — следующие 250, и так далее.

3. Синхронизация: После того как все потоки завершат свою работу, они синхронизируются с помощью директивы #pragma omp barrier. Это гарантирует, что все потоки завершили свою часть работы, прежде чем программа перейдет к следующему шагу.

3. Далее проведем анализ для различной размерности.

	A	B	C
1	Размерность	последовательно	параллельно
2	10	0.0000002	0.001549
3	100	0.0000003	0.000854
4	1000	0.0000027	0.000983
5	10000	0.0000148	0.002199
6	100000	0.0001875	0.001249
7	1000000	0.0022274	0.004748
8	10000000	0.0268678	0.039189
9	100000000	0.2122303	0.264341

Параллельная версия программы начинает проявлять преимущество при больших объемах данных. Начиная с размерности порядка 1000, она работает быстрее последовательной. Для больших массивов данных (от 1 миллиона и выше) параллельная версия демонстрирует значительное ускорение.

Для маленьких задач (например, до 1000 элементов), накладные расходы на создание и управление потоками в параллельной версии могут превысить выгоды от параллелизма, из-за чего последовательная версия оказывается более эффективной.

Листинг программ

Последовательная версия

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <ctime>  // Для измерения времени
#include <omp.h>
#include "windows.h"

struct Point {
   double x, y, z;
};

void transform_points(std::vector<Point>& points, double offset_x, double offset_y, double offset_z) {
   for (auto& point : points) {
      point.x += offset_x;
      point.y += offset_y;
      point.z += offset_z;
   }
}

int main() {
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
   const int num_points = 100; // Количество точек
   std::vector<Point> points(num_points); // Создание вектора точек


   for (int i = 0; i < num_points; ++i) {
            points[i] = { double(i), double(i), double(i) }; // Каждая точка инициализируется значениями i
   }

   double offset_x = 1.0, offset_y = 2.0, offset_z = 3.0;

   double start_time = omp_get_wtime(); // Засекаем начальное время
```

```
    transform_points(points, offset_x, offset_y, offset_z);

    double end_time = omp_get_wtime(); // Засекаем конечное время

     double elapsed_time = double(end_time - start_time);// Время выполнения в секундах

       std::cout << "Sequential Time: " << std::fixed << std::setprecision(6) << elapsed_time << " seconds\n";

    return 0;
}
```

Параллельная версия

```
#include <iostream>
#include <vector>
#include <omp.h>
#define NUM_THREADS 8

struct Point {
    double x, y, z;
};


void transform_points_parallel(std::vector<Point>& points, double offset_x, double offset_y, double offset_z) {
    int num_points = points.size();
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for
    for (int i = 0; i < num_points; ++i) {
        points[i].x += offset_x;
        points[i].y += offset_y;
        points[i].z += offset_z;
    }
#pragma omp barrier
}
```

```
int main() {
    const int num_points = 1000000;
    std::vector<Point> points(num_points);

    for (int i = 0; i < num_points; ++i) {
                points[i] = { double(i), double(i), double(i) }; // Каждая точка инициализируется значениями i
    }


    double offset_x = 1.0, offset_y = 2.0, offset_z = 3.0;


    double start_time = omp_get_wtime(); // Засекаем начальное время

    transform_points_parallel(points, offset_x, offset_y, offset_z);

    double end_time = omp_get_wtime(); // Засекаем конечное время


    double elapsed_time = end_time - start_time; // Время выполнения в секундах

    std::cout << "Parallel Time: " << elapsed_time << " seconds\n";

    return 0;
}
```