МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №2

по дисциплине: Параллельное программирование
тема: «Реализация параллелизма в рамках стандарта OpenMP»

Выполнил: ст. группы ПВ-223
Игнатьев Артур Олегович

Проверили:
доц. Островский Алексей Мичеславович

Белгород 2025 г.

Лабораторная работа №2

Реализация параллелизма в рамках стандарта OpenMP.

Цель работы: Изучить возможности стандарта OpenMP для создания многопоточных программ, изучить механизмы управления потоками и различные стратегии распределения работы между потоками, их влияние на производительность вычислений.

Цель работы обуславливает постановку и решение следующих задач:

1. Ознакомиться с основами OpenMP, включая компиляцию и запуск многопоточных программ.
2. Изучить работу командной модели потоков OpenMP.
3. Рассмотреть основы параллельного вычисления в цикле с разными стратегиями распределения работы между потоками (static, dynamic, guided, auto).
4. Выполнить индивидуальное задание, связанное с использованием стандарта OpenMP для реализации вычислительной задачи. Следует декомпозировать вычислительную задачу, вычленив сущности для потоковой обработки. **Внимание! В процессе декомпозиции вычленяйте сущности различным образом, с тем чтобы рассмотреть стратегии распределения работ между потоками в условиях пропорциональных и диспропорциональных вычислительных нагрузок.**
5. Провести анализ эффективности каждой стратегии, применительно к поставленной задаче, измеряя ускорение и масштабируемость решения при увеличении числа потоков. Описать наблюдения, сделав выводы о влиянии различных стратегий балансировки нагрузки на производительность вычислений.

Индивидуальное задание

Вариант 3

$$S = \sum_{i=1}^{N} \frac{(i+1)! + \sin^2(i) \cdot e^{-i}}{(2i+1)^2 + \cos(i)}$$

Ход выполнения лабораторной работы

Файл lab2_1.c

```c
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <omp.h>

#define EPSILON 1e-100  // Малое число для предотвращения деления на 0

// Куммер-суммирование для минимизации ошибок округления
void kahan_sum(double* sum, double term, double* comp) {
    double y = term - *comp;
    double t = *sum + y;
    *comp = (t - *sum) - y;
    *sum = t;
}

// Функция вычисления одного слагаемого
double term(int i) {
    double log_factorial = lgamma(i + 2);
    double sin_part = pow(sin(i), 2);
    double exp_part = exp(-i);

    // Числитель (логарифм)
    double log_numerator = log_factorial;
    if (sin_part * exp_part > EPSILON) {
        log_numerator = log(exp(log_factorial) + sin_part * exp_part);
    }

    // Знаменатель (логарифм)
    double denominator = pow(2 * i + 1, 2) + cos(i) + EPSILON;

    return exp(log_numerator - log(denominator));
}

// Функция вычисления суммы с OpenMP + Kahan summation
void compute_sum(FILE* file, int N, int num_threads, omp_sched_t schedule_type, int
chunk_size) {
    double S = 0.0;
    double compensation = 0.0;
    double start_time, end_time;

    omp_set_num_threads(num_threads);
    omp_set_schedule(schedule_type, chunk_size);

    start_time = omp_get_wtime();

    #pragma omp parallel
    {
        double local_S = 0.0;
        double local_comp = 0.0;

        #pragma omp for schedule(runtime) nowait
```

```
        for (int i = 1; i <= N; i++) {
            double t = term(i);
            if (!isinf(t) && !isnan(t)) {
                kahan_sum(&local_S, t, &local_comp);
            }
        }

        #pragma omp atomic
        S += local_S;
    }

    end_time = omp_get_wtime();

    // Определяем название стратегии
    const char* schedule_name =
        (schedule_type == omp_sched_static) ? "static" :
        (schedule_type == omp_sched_dynamic) ? "dynamic" :
        (schedule_type == omp_sched_guided) ? "guided" : "auto";

    // Вывод в консоль
    printf("Threads: %d, Schedule: %s, Time: %f sec, S = %.10e\n",
           num_threads, schedule_name, end_time - start_time, S);

    // Запись в CSV
    fprintf(file, "%d,%s,%f,%.10e\n", num_threads, schedule_name, end_time - start_time, S);
}

int main() {
    int N = 1000000;
    int num_threads_list[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18};
    int chunk_size = 100;

    printf("Computing sum for N = %d with different scheduling strategies\n", N);

    // Открываем файл для записи
    FILE* file = fopen("results.csv", "w");
    if (!file) {
        perror("Ошибка открытия файла");
        return 1;
    }

    // Записываем заголовок CSV
    fprintf(file, "Threads,Schedule,Time,S\n");

    for (int i = 0; i < 18; i++) {
        compute_sum(file, N, num_threads_list[i], omp_sched_static, chunk_size);
        compute_sum(file, N, num_threads_list[i], omp_sched_dynamic, chunk_size);
        compute_sum(file, N, num_threads_list[i], omp_sched_guided, chunk_size);
        compute_sum(file, N, num_threads_list[i], omp_sched_auto, chunk_size);
    }

    // Закрываем файл
    fclose(file);

    return 0;
```

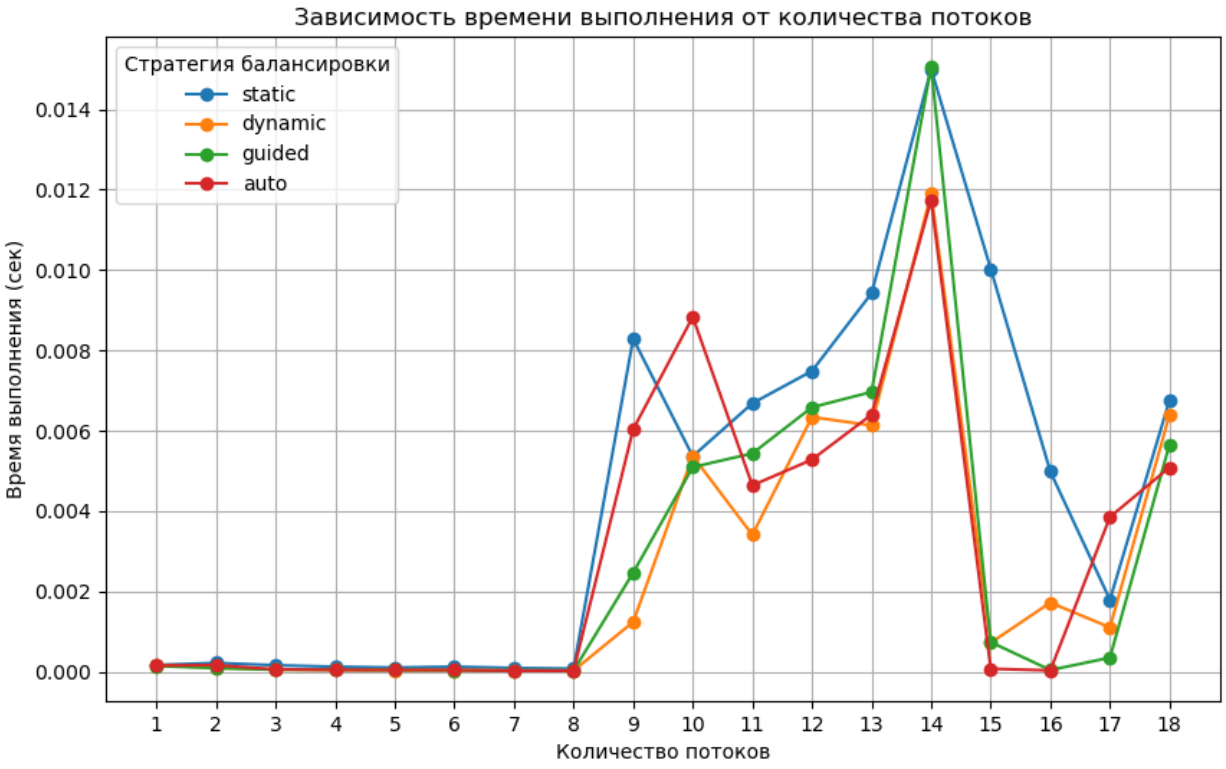
```
}
```

Файл results.py

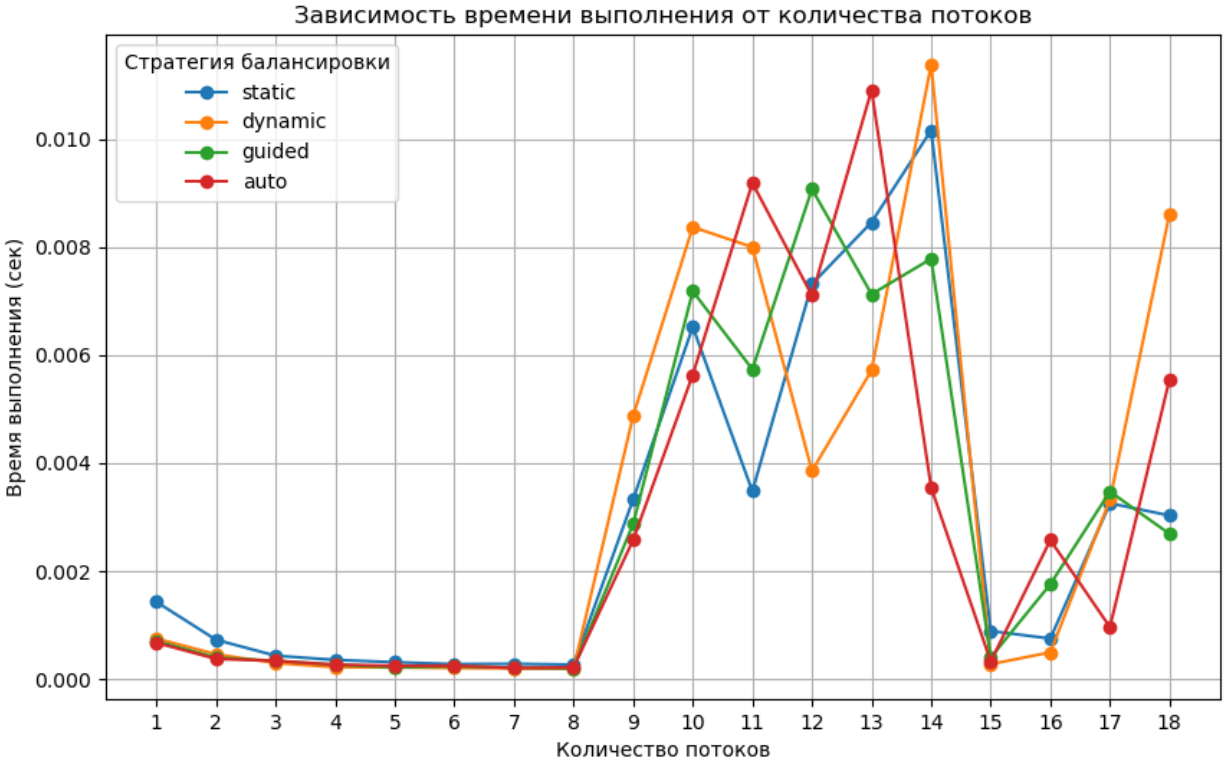
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Загружаем данные из CSV
csv_filename = "results.csv"
df = pd.read_csv(csv_filename)

# Список уникальных стратегий балансировки
schedules = df["Schedule"].unique()

# Определяем уникальные значения потоков (X-ось)
threads_unique = sorted(df["Threads"].unique())

# Создаём график
plt.figure(figsize=(10, 6))

# Построение линий для каждой стратегии
for schedule in schedules:
    subset = df[df["Schedule"] == schedule]
    plt.plot(subset["Threads"], subset["Time"], marker="o", linestyle="-", label=schedule)

# Настройки графика
plt.xlabel("Количество потоков")
plt.ylabel("Время выполнения (сек)")
plt.title("Зависимость времени выполнения от количества потоков")
plt.legend(title="Стратегия балансировки")
plt.grid(True)

# Настройка оси X для отображения только целых значений потоков
plt.xticks(threads_unique, [str(int(t)) for t in threads_unique])

# Сохраняем график
plt.savefig("execution_time_plot.png")
plt.show()
```

Результаты

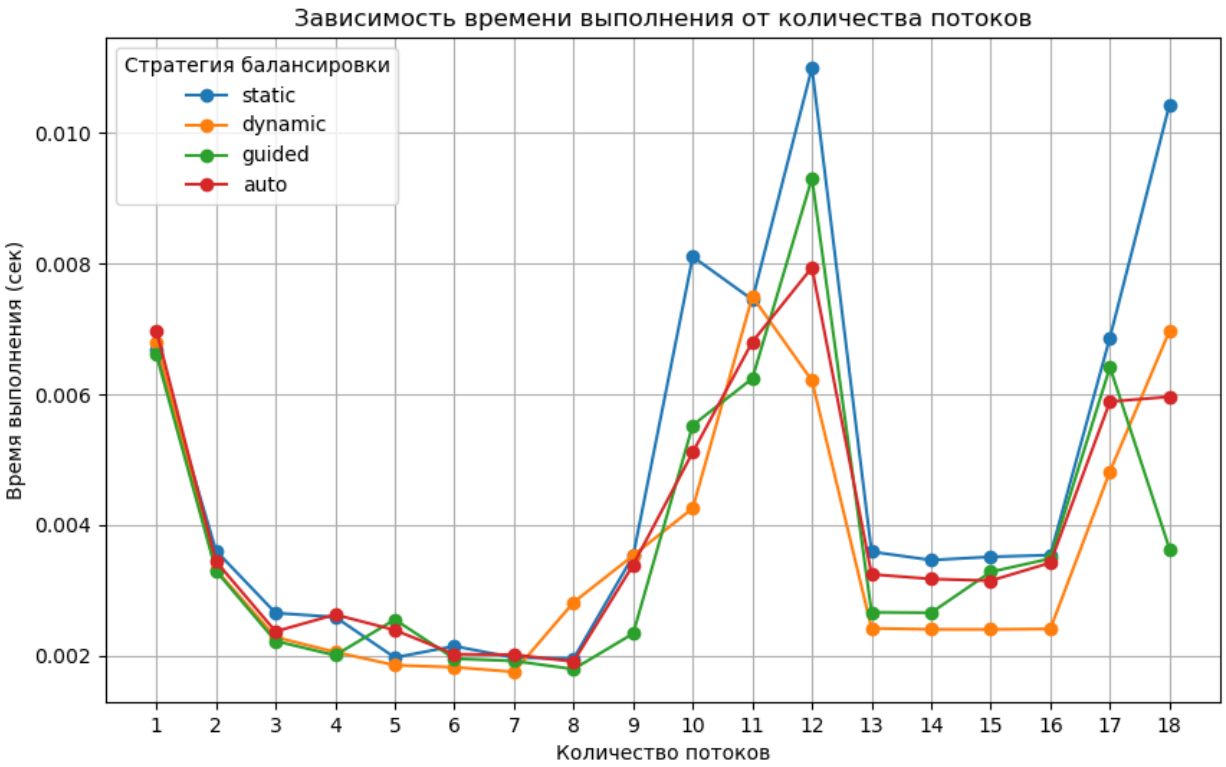
N = 1000



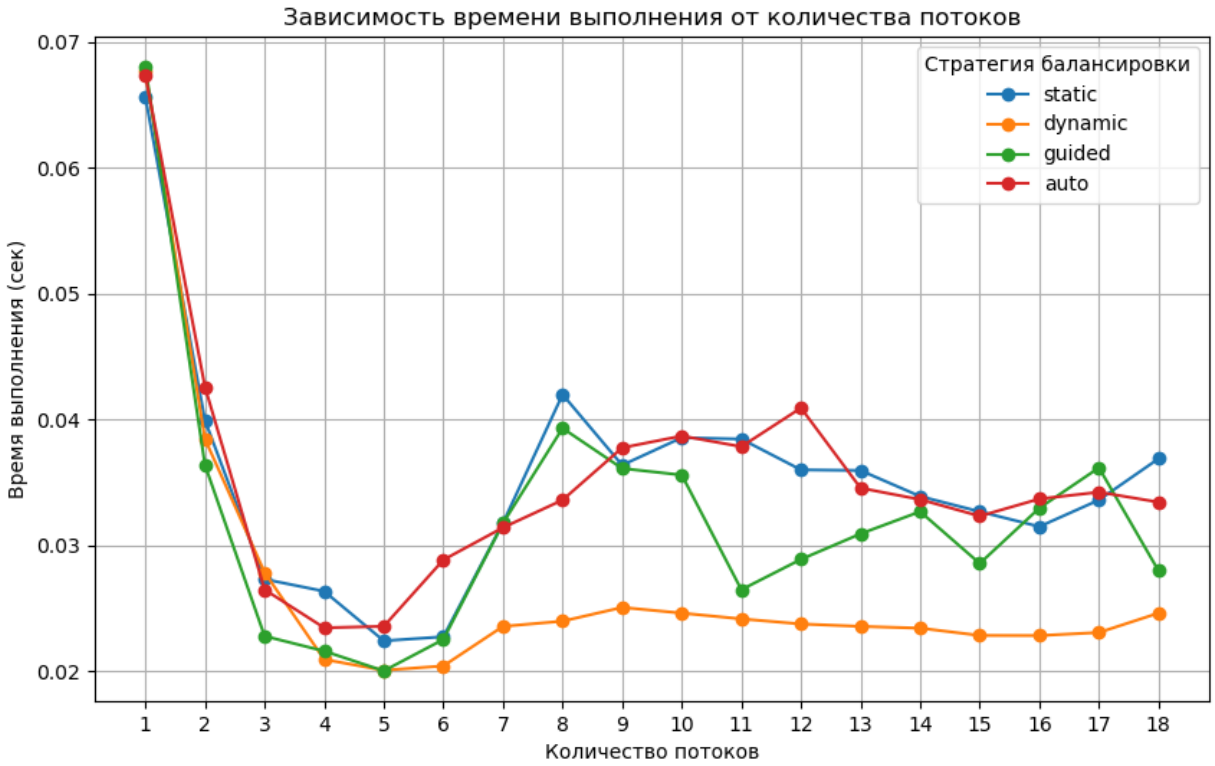
N = 10000



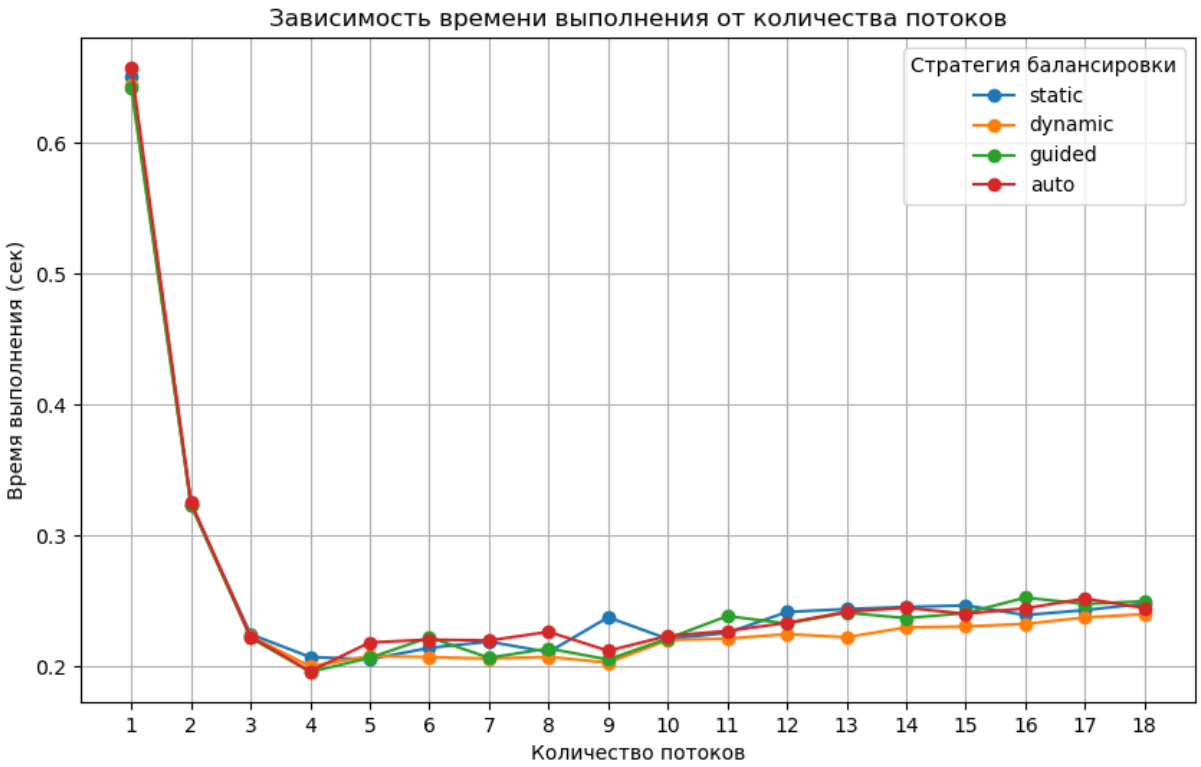
N = 100000



N = 1000000



N = 10000000



Выводы работы программы от количества членов ряда

Для правильного вывода, стоит сначала разобрать какие ядра используются. Это 4P + 8E+2L.

P-ядра высокопроизводительные, имеют многопоточность и работают на тактовой частоте до 4.5GHz.

E-ядра энергоэффективные, работают на более низкой тактовой частоте до 3.6GHz и не имеют многопоточности.

L-ядра низкого энергопотребления, работают на тактовой частоте до 2.5GHz, обычно не используются в системе и нужны для работы в условиях сна.

Немного о методах распределения

При статическом (static) распределении итераций:

- Все итерации делятся на равные блоки (по chunk_size) и назначаются потокам заранее.
- Если chunk_size не указан, количество итераций делится поровну между потоками.
- Хорошо работает, если все итерации выполняются примерно за одинаковое время.

При динамическом (dynamic) распределении:

- Изначально никто не получает итерации заранее.

- Когда поток освобождается, он берет следующий доступный блок chunk_size итераций.
- Подходит для случаев, когда время выполнения итераций непредсказуемо.

При управляемом (guided) распределении:

- Потоки получают крупные блоки в начале и меньшие по мере выполнения.
- OpenMP автоматически уменьшает размер chunk_size (по умолчанию 1).
- Отлично работает, если начальные итерации требуют больше вычислений.

При автоматическом (auto) распределении:

- OpenMP сам решает, как лучше разделить итерации (на основе профилирования).
- Неявный аналог других стратегий, но без явного указания способа разделения.

Тестирование проводилось в Ubuntu WSL(Windows Subsystem for Linux), а значит планировщик поток используется из Windows. Это даёт нам ясность в распределении нагрузки на потоки. Windows предпочитает использовать E-ядра для не высокой нагрузки, что в свою очередь увеличивает время выполнения из-за более низкой тактовой частоты.

На графиках видно, что вычисление при использовании до 8 потоков выполняется быстрее, так как используются P-ядра и их тактовая частота обычно значительно поднимается от любых нагрузках, однако при использовании потоков выше 8, основное вычисление начинает происходить на E-ядрах и время выполнения увеличивается. При использовании большего числа E ядер и количества вычислений, тактовая частота повышается больше, что и даёт прирост на графике где N = 100000.
Можно отметить, что в общем наибольший прирост получаем при добавления 2го потока. Дальше прирост есть, но уже не такой ощутимый.

На графиках видно, что лучше всего справляется dynamic, так как вычислений много, но они не трудозатратные, поэтому чем быстрее поток начнёт производить вычисления, тем быстрее будет происходить следующее вычисление.

Вывод: на этой лабораторной работе мы ознакомились с основами OpenMP, включая компиляцию и запуск многопоточных программ. Изучили работу командной модели потоков OpenMP. Рассмотрели основы параллельного вычисления в цикле с разными

стратегиями распределения работы между потоками (static, dynamic, guided, auto). Выполнили индивидуальное задание, связанное с использованием стандарта OpenMP для реализации вычислительной задачи.