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
Пахомов Владислав Андреевич

Проверили:
доц. Островский Алексей Мичеславович

Белгород 2025 г.

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.

Условие индивидуального задания:

$$S = \sum_{i=1}^{N} \frac{cos(i^3)+i^4e^{-i}+ln(i+1)}{\sqrt{i^2+tan(i)+1}+i!}$$

Ход выполнения работы

Рассмотрим задачу при **пропорциональной нагрузке**:

В отличие от прошлого задания, самая тяжёлая задача, которую можно выполнить только синхронно (а именно расчёт факториалов) была выполнена перед запуском кода, что позволило составить таблицу факториалов.

Исходный код:

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

#define NUM_ITERATIONS 10000000

int main()
{
    // Непараллельные вычисления факториалов
    long double *factorials = malloc(NUM_ITERATIONS * sizeof(long double));
    factorials[0] = 1;

    for (int i = 1; i < NUM_ITERATIONS; i++)
    {
        factorials[i] = factorials[i - 1] * (i + 1.0);
    }

    long double *sums = malloc(NUM_ITERATIONS * sizeof(long double));
    long double sum = 0.0;

#pragma omp parallel
    {
#pragma omp for schedule(auto)
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
            sums[i] = (cos(pow(i, 3)) + pow(i, 4) * exp(-i) + log(i + 1)) /
                      (sqrt(i * i + tan(i) + 1) + factorials[i]);
        }

#pragma omp for reduction(+ : sum)
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
            sum += sums[i];
        }
    }
```

```
    free(factorials);
    free(sums);

    return 0;
}
```

Результаты вычислений по времени:

Стратегия	static	dynamic	auto	guided
Время выполнения первый запуск, с	1.342	1.513	1.376	1.323
Время выполнения после прогрева, с	1.327	1.506	1.349	1.291
Прирост после прогрева, %	1.130	0.465	2.001	2.479

Рассмотрим задачу при **непропорциональной нагрузке**:

Здесь будем вычислять факториал в каждой итерации в отличие от прошлого подхода, чем больше итерация, тем дольше будет выполняться программа:

Исходный код:

```
#include <stdio.h>
#include <omp.h>
#include <math.h>
#include <stdlib.h>

#define NUM_ITERATIONS 10000

long double factorial(const int n)
{
    long double f = 1;
    for (int i = 1; i <= n; ++i)
        f *= i;
    return f;
}

int main()
{
    long double *sums = malloc(NUM_ITERATIONS * sizeof(long double));
    long double sum = 0.0;

#pragma omp parallel
    {
#pragma omp for schedule(guided)
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
            sums[i] = (cos(pow(i, 3)) + pow(i, 4) * exp(-i) + log(i + 1)) /
                      (sqrt(i * i + tan(i) + 1) + factorial(i + 1));
        }

#pragma omp for reduction(+ : sum)
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
            sum += sums[i];
```

```
        }
    }

    free(sums);

    return 0;
}
```

Результаты вычислений по времени:

Стратегия	static	dynamic	auto	guided
Время выполнения первый запуск, с	0.484	0.259	0.502	0.249
Время выполнения после прогрева, с	0.481	0.246	0.478	0.248
Прирост после прогрева, %	0.624	5.285	5.021	0.403

Попробуем декомпозировать задачу по-другому:

В отдельных итерациях будем считать числитель, знаменатель, результат их деления и потом - сумму. Здесь попытаемся увеличить скорость программы за счёт прогрева кеша путём декомпозиции задачи на более простые:

Исходный код:

```
#include <stdio.h>
#include <omp.h>
#include <math.h>
#include <stdlib.h>

#define NUM_ITERATIONS 20000

long double factorial(const int n)
{
    long double f = 1;
    for (int i = 1; i <= n; ++i)
        f *= i;
    return f;
}

int main()
{
    long double *sums = malloc(NUM_ITERATIONS * sizeof(long double));
    long double *numerator = malloc(NUM_ITERATIONS * sizeof(long double));
    long double *denumerator = malloc(NUM_ITERATIONS * sizeof(long double));
    long double sum = 0.0;

#pragma omp parallel
    {
#pragma omp for schedule(static)
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
            numerator[i] = cos(pow(i, 3)) + pow(i, 4) * exp(-i) + log(i + 1);
        }
#pragma omp for schedule(guided)
        for (int i = 0; i < NUM_ITERATIONS; i++)
```

```
        {
            denumerator[i] = sqrt(i * i + tan(i) + 1) + factorial(i + 1);
        }
#pragma omp for schedule(static)
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
            sums[i] = numerator[i] / denumerator[i];
        }

#pragma omp for reduction(+ : sum)
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
            sum += sums[i];
        }
    }

    free(sums);

    return 0;
}
```

Результаты вычислений по времени:

Стратегия	static	dynamic	auto	guided	Разные стратегии
Время выполнения первый запуск, с	2.137	1.179	2.190	1.187	1.182
Время выполнения после прогрева, с	2.098	1.169	2.105	1.179	1.173
Прирост после прогрева, %	1.858	0.855	4.038	0.679	0.767

Вывод: в ходе лабораторной работы изучили возможности стандарта OpenMP для создания многопоточных программ, изучить механизмы управления потоками и различные стратегии распределения работы между потоками, их влияние на производительность вычислений. Рассмотрим пропорциональную нагрузку. Оптимизация вычислений и вынос недекомпозируемой задачи и её кеширование в дальнейших вычислениях позволило сократить время выполнения программы, в лабораторной работе №1 было достигнуто время в 1.5 секунды, в то время как минимальное время при новом подходе 1.3 секунды. Неожиданным оказался тот факт, что самым быстрым оказалась стратегия guided, судя по всему один из компонентов формулы имеет нелинейную сложность вычисления, следовательно и стратегия guided оказала больший эффект, так как при малых значениях программа выполняется быстрее следовательно и поток может выполнить больше итераций. Самой долгой оказалась стратегия dynamic, она здесь действительно не подходит, так как компоненты вычисления не имеют случайную стоимость вычисления по времени. Хорошо себя показал и static, что довольно логично, влияние нелинейных компонентов было невелико и следовательно время увеличилось незначительно по сравнению с guided. Стратегия auto, если судить по времени, отдала предпочтение стратегии static. В непропорциональной нагрузке сразу можно отметить меньшую производительность, получилось достигнуть лишь N = 10000 по сравнению с первым методом N = 10000000. static, ожидаемо, дала худшее время из-за ручного вычисления факториала на каждом шагу. auto всё ещё отдаёт предпочтение static, а значит так же работает очень плохо. dynamic и guided стратегии дали двухкратное уменьшение времени. dynamic болле хорошо работает с неравномерной нагрузкой с большим временем выполнения потока. guided

также даёт неплохое время, так как лучше всего подходит специфике выбранного алгоритма. Ручная настройка стратегии даёт прирост производительности на основе знаний программиста о программе и его сложности. static лучше всего подходит для задач, где время выполнения на каждой итерации постоянно. guided лучше всего подходит для задач, где время выполнения растёт экспоненциально или x^N. dynamic подойдёт для задач, в которых время выполнения итерации не зависит от номера итерации, а время выполнения каждой итерации достаточно большое, чтобы затраты на вычисление, какой из потоков ядра свободен, были относительно малыми. auto обычно работает хуже всего, тонкая настройка даёт прирост производительности. Были также проведены дополнительные эксперименты с прогревом кеша. Задача была декомпозирована на более мелкие а количество шагов было увеличено. Мы исходили из предположения, что задача будет выполняться быстрее, если задача будет декомпозирована на большее количество шагов из-за прогрева кеша. Однако гипотеза не была подтверждена на практике. Для декомпозиции в равной степени происходит ускорение программы после прогрева, как и в вариантах программы без декомпозиции. Возможно, такой результат был получен вследствие того что было проведено недостаточно шагов декомпозиции. Необходимо проводить дальнейшие эксперименты для более сложной для вычисления задачи.