**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №3**

**по дисциплине**

**«Операционные системы»**

Тема: **Процессы и потоки**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3316 |  | Кудрявцев А.А. |
| Преподаватель |  | Тимофеев А.В. |

Санкт-Петербург

2025

**Задание 3.2. Реализация многопоточного приложения с использованием технологии OpenMP.**

**Цель работы**: исследовать механизмы создания и управления процессами и потоками в ОС Windows.

**Описание алгоритма:**

Для приближённого вычисления числа π используется метод прямоугольников (интегральное приближение):

****

**1. Глобальные и ключевые переменные:**

* **pi** — переменная для хранения финального приближённого значения числа π.
* **sum** — промежуточная сумма, в которую аккумулируются значения из каждого потока с помощью механизма reduction.
* **delta** —1.0 / N, используется в формуле для вычисления.

**2. Константы:**

* **N = 10 000 000** — общее количество итераций, определяет точность расчёта числа π.
* **BLOCK\_SIZE = 3316130** — размер блока итераций (на основе номера студенческого билета).
* **NUM\_THREADS = 4** — количество потоков, используемых в расчёте.

**3. Основной алгоритм:**

* Сначала устанавливается количество потоков с помощью функции omp\_set\_num\_threads().
* Время выполнения измеряется функциями omp\_get\_wtime() до и после выполнения основного параллельного блока.
* Главный расчёт числа π осуществляется в директиве OpenMP

#pragma omp parallel for:

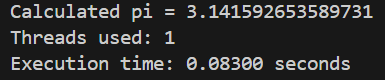
* + Параллельный цикл выполняется всеми потоками одновременно.
  + Используется директива reduction(+:sum), которая позволяет всем потокам безопасно суммировать результат в одну переменную sum без гонки данных.
  + Распределение итераций между потоками происходит динамически с шагом BLOCK\_SIZE, что обеспечивает гибкую загрузку потоков.

**4. Особенности реализации:**

* OpenMP автоматически обрабатывает создание, синхронизацию и завершение потоков.
* Нет необходимости явно использовать mutex или WaitForSingleObject как в случае с WinAPI — OpenMP управляет доступом к переменным через reduction.
* Использование schedule(dynamic, BLOCK\_SIZE) позволяет лучше сбалансировать нагрузку при неравномерной сложности вычислений на разных итерациях (в данном случае — просто демонстрация).
* parallel for - создает параллельную версию цикла for
* reduction(+:sum) - обеспечивает корректное суммирование результатов из разных потоков
* schedule(dynamic, BLOCK\_SIZE) - динамически распределяет итерации блоками указанного размера

|  |  |
| --- | --- |
| Количество потоков | Время выполнения |
| 1 | 0,083 |
| 2 | 0,061 |
| 4 | 0,037 |
| 8 | 0,041 |
| 12 | 0,039 |
| 16 | 0,045 |

**Пример выполнения программы**



**Текст программы**

#include <stdio.h>

#include <omp.h>

#include <time.h>

#define N 10000000

#define BLOCK\_SIZE 3316130

#define NUM\_THREADS 16

int main() {

    omp\_set\_num\_threads(NUM\_THREADS);

    double pi = 0.0;

    double sum = 0.0;

    const double delta = 1.0 / N;

    double start\_time = omp\_get\_wtime();

    #pragma omp parallel for reduction(+:sum) schedule(dynamic, BLOCK\_SIZE)

    for (int i = 0; i < N; i++) {

        double x = (i + 0.5) \* delta;

        sum += 4.0 / (1.0 + x \* x);

    }

    pi = sum \* delta;

    double end\_time = omp\_get\_wtime();

    double elapsed\_time = end\_time - start\_time;

    printf("Calculated pi = %.15f\n", pi);

    printf("Threads used: %d\n", NUM\_THREADS);

    printf("Execution time: %.5f seconds\n", elapsed\_time);

    return 0;

}

**Выводы**

В данной работе реализовано эффективное многопоточное вычисление числа π с использованием низкоуровневых средств Open NP. Программа демонстрирует базовые приёмы работы с потоками, синхронизацией и точными таймерами, что важно для системного программирования. Методика может применяться для решения широкого класса задач с большим количеством итеративных операций.

**Анализ результатов:**

* **При увеличении числа потоков от 1 до 4** наблюдается **заметное ускорение** выполнения. Это говорит о том, что задача хорошо масштабируется на многопроцессорных системах **до определённого предела**.
* **После 4 потоков** ускорение **перестаёт расти**, а время даже **незначительно увеличивается** (особенно на 16 потоках). Это связано с:
* Перегрузкой потоками, если физически ядер меньше, чем потоков.
* Повышенными накладными расходами на управление потоками.
* Переключением контекста между потоками.
* **Оптимальное число потоков — 4 или 8** (в зависимости от архитектуры процессора).
* **Параллельная реализация с OpenMP оказалась эффективной**, особенно в сравнении с однопоточной реализацией — в 2–2.5 раза быстрее при 4 потоках.