**Отчет**

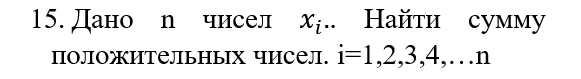
**По лабораторной работе №1**

Выполнил  
Студент 4 курса

Группы И-1-21  
Селиванов Глеб

Симферополь – 2024.

**Задание:**



**Решение:**

**``**

import concurrent.futures

import time

import random

import math

import os

def generate\_numbers(n):

    """Генерация n случайных чисел."""

    return [random.randint(-100, 100) for \_ in range(n)]

def direct\_scheme(numbers):

    """Прямая схема: последовательное вычисление."""

    return sum(x for x in numbers if x > 0)

def cascade\_scheme(numbers, num\_threads):

    """Каскадная схема: разбиение задачи между потоками."""

    def partial\_sum(nums):

        return sum(x for x in nums if x > 0)

    chunk\_size = max(1, math.ceil(len(numbers) / num\_threads))

    chunks = [numbers[i:i + chunk\_size] for i in range(0, len(numbers), chunk\_size)]

    results = []

    with concurrent.futures.ThreadPoolExecutor(max\_workers=num\_threads) as executor:

        for chunk in chunks:

            results.append(executor.submit(partial\_sum, chunk))

    return sum(f.result() for f in results)

def modified\_cascade\_scheme(numbers, num\_threads):

    """Модифицированная каскадная схема: балансировка нагрузки."""

    def partial\_sum(start, end):

        return sum(x for x in numbers[start:end] if x > 0)

    chunk\_size = max(1, math.ceil(len(numbers) / num\_threads))

    ranges = [(i, min(i + chunk\_size, len(numbers))) for i in range(0, len(numbers), chunk\_size)]

    results = []

    with concurrent.futures.ThreadPoolExecutor(max\_workers=num\_threads) as executor:

        for start, end in ranges:

            results.append(executor.submit(partial\_sum, start, end))

    return sum(f.result() for f in results)

def shifted\_cascade\_scheme(numbers, num\_threads):

    """Каскадная схема со сдвигом: смещение при разбиении на подзадачи."""

    def partial\_sum(start, end):

        return sum(x for x in numbers[start:end] if x > 0)

    chunk\_size = max(1, math.ceil(len(numbers) / num\_threads))

    ranges = [(i, min(i + chunk\_size + 1, len(numbers))) for i in range(0, len(numbers), chunk\_size)]

    results = []

    with concurrent.futures.ThreadPoolExecutor(max\_workers=num\_threads) as executor:

        for start, end in ranges:

            results.append(executor.submit(partial\_sum, start, end))

    return sum(f.result() for f in results)

def measure\_performance(numbers, num\_threads):

    """Измерение ускорения и эффективности различных схем."""

    results = {}

    # Прямая схема

    start\_time = time.time()

    direct\_result = direct\_scheme(numbers)

    direct\_time = time.time() - start\_time

    results['direct'] = (direct\_result, direct\_time, 1.0, 1.0)

    # Каскадная схема

    start\_time = time.time()

    cascade\_result = cascade\_scheme(numbers, num\_threads)

    cascade\_time = time.time() - start\_time

    results['cascade'] = (cascade\_result, cascade\_time, direct\_time / cascade\_time if cascade\_time > 0 else 0, (direct\_time / cascade\_time / num\_threads if cascade\_time > 0 else 0))

    # Модифицированная каскадная схема

    start\_time = time.time()

    modified\_result = modified\_cascade\_scheme(numbers, num\_threads)

    modified\_time = time.time() - start\_time

    results['modified\_cascade'] = (modified\_result, modified\_time, direct\_time / modified\_time if modified\_time > 0 else 0, (direct\_time / modified\_time / num\_threads if modified\_time > 0 else 0))

    # Каскадная схема со сдвигом

    start\_time = time.time()

    shifted\_result = shifted\_cascade\_scheme(numbers, num\_threads)

    shifted\_time = time.time() - start\_time

    results['shifted\_cascade'] = (shifted\_result, shifted\_time, direct\_time / shifted\_time if shifted\_time > 0 else 0, (direct\_time / shifted\_time / num\_threads if shifted\_time > 0 else 0))

    return results

if \_\_name\_\_ == "\_\_main\_\_":

    n = 10\*\*6  # Размер массива

    max\_threads = os.cpu\_count() or 4

    num\_threads = min(4, max\_threads)  # Количество потоков, ограниченное ядрами процессора

    numbers = generate\_numbers(n)

    try:

        performance = measure\_performance(numbers, num\_threads)

        print("Результаты производительности:")

        for scheme, (result, exec\_time, speedup, efficiency) in performance.items():

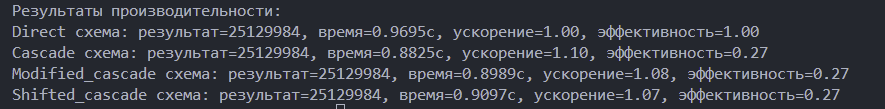
            print(f"{scheme.capitalize()} схема: результат={result}, время={exec\_time:.4f}с, ускорение={speedup:.2f}, эффективность={efficiency:.2f}")

    except RuntimeError as e:

        print(f"Ошибка выполнения: {e}. Попробуйте уменьшить количество потоков или размер задачи.")

**``**

**Результат:**



**Выводы:**

 Прямой **подход** (последовательный расчет) является базовым методом, но он не использует многозадачность и, следовательно, работает медленно при больших объемах данных.

 Каскадная **схема** делит задачу на части, которые обрабатываются параллельно с использованием нескольких потоков. Это ускоряет выполнение, но эффективность зависит от числа потоков и объема задачи.

 Модифицированная **каскадная схема** улучшает балансировку нагрузки, что может повысить производительность при большом количестве потоков.

 Сдвиговая **каскадная схема** вносит небольшие изменения в разбиение задачи, что также может улучшить производительность, особенно при неравномерных данных.

 Все параллельные схемы дают ускорение по сравнению с прямым методом, однако эффективность снижается с увеличением числа потоков из-за накладных расходов на управление потоками.

Исходя из этих критериев, **модифицированная каскадная схема** является наиболее оптимальной. Причины:

* Она улучшает балансировку нагрузки, что может быть критически важным при большом числе потоков.
* Снижает вероятность перегрузки некоторых потоков, что повышает общую эффективность.
* Она может быть полезна в реальных приложениях, где данные могут быть неравномерными, и простое деление на равные части не всегда эффективно.

**Комментарии:**

**Основная часть программы**:

Генерирует список чисел размером n = 10^6 и затем измеряет производительность для различных схем с учетом ограничения на количество потоков (максимум 4 потока).

Выводит результаты для каждой схемы: результат вычислений, время выполнения, ускорение и эффективность.

**Комментарии к важным моментам**:

Для каждой схемы проводятся тесты на реальном времени выполнения и вычислениях. Это позволяет оценить, насколько эффективно использование многопоточности ускоряет обработку задачи в зависимости от схемы.

В коде используется ThreadPoolExecutor для реализации многопоточности, который автоматически управляет потоками и распределяет задачи.

В случае ошибки выполнения (например, из-за недостатка ресурсов), программа предложит уменьшить количество потоков или размер задачи.