

Міністерство освіти і науки України
Національний університет «Львівська політехніка»
Кафедра автоматизованих систем управління



Звіт
до лабораторної роботи №1
« Способи розпаралелювання та організації обчислень. Послідовні
алгоритми. »
з дисципліни
« Паралельні обчислення і розподілені системи »

Виконала: студентка групи ОІ-32
Зелінська Єлизавета

Прийняла: Бадзь В.М.

Львів – 2025

Лабораторна робота №1

Мета роботи: Оволодіти практичними прийомами розробки алгоритмів та програм із застосуванням ітерації.

Послідовність роботи:

1. Використовуючи послідовні алгоритми, написати програму розв'язання індивідуального завдання.

2. При створенні програми намагатись забезпечити якнайбільшу незалежність

програмного коду від операційної системи та середовища програмування.

3. Передбачити можливості:

- формування вхідних даних заданого розміру, наприклад, за допомогою генератора випадкових чисел;

- збереження вхідних даних у файлі із заданою назвою;

- зчитування вхідних даних із заданого файлу;

- виведення результатів на екран або у файл.

4. Відлагодити програму на прикладі з невеликим об'ємом вхідних даних, результати для якого можуть бути перевірені перерахунком поза програмою.

5. Підготувати приклад вхідних даних, для якого час на розв'язання задачі складатиме приблизно 5 секунд і перевірити програму на цьому прикладі.

6. Визначити часові характеристики роботи програми (сумарний час на виконання

обчислень, не враховуючи формування вхідних даних, введення та виведення).

7. Розв'язати те ж саме завдання з використанням паралельних обчислень (multithreading). Повторити пункт 6 для цього варіанту програми.

Індивідуальне завдання:

8. Обчислити суму $1/1 + 1/2 + 1/3 + \dots$ до заданої кількості знаків після коми.

Текст програми:

```
import time
import multiprocessing
import random
from decimal import Decimal, getcontext
from concurrent.futures import ThreadPoolExecutor

def harmonic_sum_sequential(n, precision):
    getcontext().prec = precision + 5
    total = Decimal(0)
    for i in range(1, n + 1):
        total += Decimal(1) / Decimal(i)
    return round(total, precision)

def harmonic_partial(start, end, precision):
    getcontext().prec = precision + 5
    partial_sum = Decimal(0)
    for i in range(start, end + 1):
        partial_sum += Decimal(1) / Decimal(i)
    return partial_sum

def harmonic_sum_parallel(n, precision):
    getcontext().prec = precision + 5
    num_processes = min(multiprocessing.cpu_count(), n // 1000) if n > 10000 else 1
    num_processes = max(1, num_processes)

    with multiprocessing.Pool(num_processes) as pool:
        step = n // num_processes
        ranges = [(i * step + 1, (i + 1) * step) if i < num_processes - 1 else n, precision)
        for i in range(num_processes)]
        partial_sums = pool.starmap(harmonic_partial, ranges)

    return round(sum(partial_sums), precision), num_processes
```

```

def harmonic_sum_threads(n, precision):
    getcontext().prec = precision + 5
    max_threads = min(n // 500, multiprocessing.cpu_count()) if n > 10000 else 1
    num_threads = max(1, max_threads)

    step = n // num_threads
    ranges = [(i * step + 1, (i + 1) * step if i < num_threads - 1 else n, precision) for
i in range(num_threads)]

    with ThreadPoolExecutor(max_workers=num_threads) as executor:
        partial_sums = list(executor.map(lambda args: harmonic_partial(*args),
ranges))

    return round(sum(partial_sums), precision), num_threads


def read_input_from_file(filename):
    try:
        with open(filename, "r") as f:
            data = f.readlines()
            n = int(data[0].strip().split("=")[1])
            precision = int(data[1].strip().split("=")[1])
            return n, precision
    except Exception as e:
        print(f"Помилка читання файлу: {e}")
        return None, None


def generate_random_input():
    n = random.randint(100, 10000)
    precision = random.randint(2, 10)
    print(f"Згенеровано випадкові значення: n={n}, precision={precision}")
    return n, precision


def main():
    choice = input("Виберіть спосіб введення даних (1 - вручну, 2 - з файлу
input.txt, 3 - випадково): ")

```

```
if choice == "2":
    n, precision = read_input_from_file("input.txt")
    if n is None or precision is None:
        return
    elif choice == "3":
        n, precision = generate_random_input()
    else:
        n = int(input("Введіть кількість членів ряду: "))
        precision = int(input("Введіть кількість знаків після коми: "))
    with open("input.txt", "w") as f:
        f.write(f"n={n}\nprecision={precision}\n")

start_seq = time.perf_counter()
result_seq = harmonic_sum_sequential(n, precision)
time_seq = time.perf_counter() - start_seq

start_par = time.perf_counter()
result_par, num_processes = harmonic_sum_parallel(n, precision)
time_par = time.perf_counter() - start_par

start_thr = time.perf_counter()
result_thr, num_threads = harmonic_sum_threads(n, precision)
time_thr = time.perf_counter() - start_thr

with open("output.txt", "w") as f:
    f.write(f"Послідовний результат: {result_seq} (час: {time_seq:.5f} с.)\n")
    f.write(
        f"Паралельний результат (multiprocessing): {result_par} (час: {time_par:.5f} с., процеси: {num_processes})\n")
    f.write(
        f"Паралельний результат (multithreading): {result_thr} (час: {time_thr:.5f} с., потоки: {num_threads})\n")

print("Результати збережені у output.txt")
print(f"Послідовний результат: {result_seq} (час: {time_seq:.5f} сек)")
print(f"Паралельний результат (multiprocessing): {result_par} (час: {time_par:.5f} с., процеси: {num_processes})")
print(f"Паралельний результат (multithreading): {result_thr} (час: {time_thr:.5f} с., потоки: {num_threads})")
```

```
if __name__ == "__main__":  
    main()
```

Контрольні приклади та результати роботи програми:

Виберіть спосіб введення даних (1 - вручну, 2 - з файлу input.txt, 3 - випадково):

Рис.1 Введення даних

```
Введіть кількість членів ряду: 6  
Введіть кількість знаків після коми: 3
```

Рис.2 Введення даних вручну

```
main.py  input.txt x  
n 10000  
precision=4
```

Рис.3 Читання даних з файлу

```
Згенеровано випадкові значення: n=6445, precision=6  
Результати збережені у output.txt  
Послідовний результат: 9.348353 (час: 0.00707 сек)  
Паралельний результат (multiprocessing): 9.348353 (час: 0.24238 с., процеси: 1)  
Паралельний результат (multithreading): 9.348353 (час: 0.00627 с., потоки: 1)
```

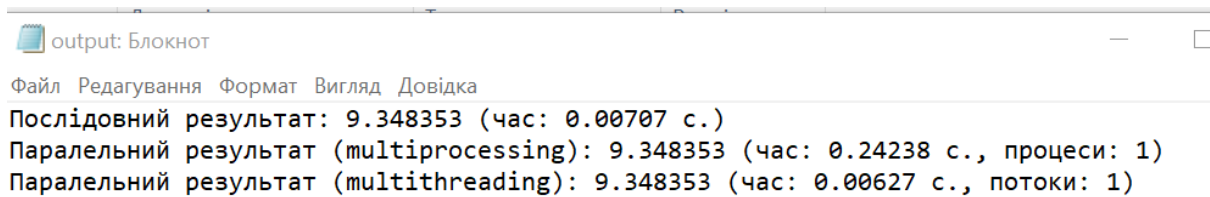
Рис. 4 Генерація даних випадковим чином

```
Послідовний результат: 2.450 (час: 0.00281 сек)  
Паралельний результат (multiprocessing): 2.450 (час: 0.39842 с., процеси: 1)  
Паралельний результат (multithreading): 2.450 (час: 0.00068 с., потоки: 1)
```

Рис.5 Результати виконання з невеликою к-тю даних

```
Послідовний результат: 9.7876 (час: 0.01011 сек)  
Паралельний результат (multiprocessing): 9.7876 (час: 0.24620 с., процеси: 1)  
Паралельний результат (multithreading): 9.7876 (час: 0.00916 с., потоки: 1)
```

Рис.6 Результати виконання з великим обсягом даних



```
output: Блокнот
Файл  Редагування  Формат  Вигляд  Довідка
Послідовний результат: 9.348353 (час: 0.00707 с.)
Паралельний результат (multiprocessing): 9.348353 (час: 0.24238 с., процеси: 1)
Паралельний результат (multithreading): 9.348353 (час: 0.00627 с., потоки: 1)
```

Рис.7 Запис результатів у файл

Висновок

У цій лабораторній роботі було досліджено три підходи до обчислення гармонічної суми: послідовний, багатопотоковий і багатопроцесний. Послідовний метод є найпростішим у реалізації, але виконується повільніше, особливо при великих значеннях n . Багатопотоковий метод, хоч і використовує декілька потоків, не дає значного прискорення через обмеження GIL у Python, що не дозволяє виконувати CPU-інтенсивні задачі одночасно в різних потоках. Найефективнішим виявився підхід із використанням процесів, який дозволяє розподілити навантаження між ядрами процесора та досягти реального паралельного виконання. Проте варто зазначити, що паралельні методи дають відчутне прискорення лише при великих обсягах даних, оскільки для малих n накладні витрати на створення потоків чи процесів можуть навіть уповільнити обчислення. Таким чином, вибір підходу залежить від розміру задачі: для невеликих обчислень ефективним залишається послідовний метод, тоді як для великих значень n доцільно використовувати багатопроцесний підхід.