Звіт по мультипоточності, а)

Маємо з лаби2 застосунок, який за допомогою HTTP-запиту до Google Books API (стандартного, за URL форматом: [https://www.googleapis.com/books/v1/volumes?q={keyword}&maxResults](https://www.googleapis.com/books/v1/volumes?q=%7bkeyword%7d&maxResults)= ) для різних ключових слів із параметром maxResults= виконує пошук книжок.

*maxResults - кількість відображених результатів для 1 запиту.*

Застосунок іноді “тормозить” через те, що пошукові запити виконуються послідовно, і це займає багато часу. Для підвищення продуктивності запропоновано використовувати мультипоточність, що дозволяє виконувати запити паралельно. Це має зменшити загальний час очікування та покращити швидкодію застосунку. У звіті буде проведено порівняння послідовного та паралельного виконання запитів, щоб оцінити реальний ефект від впровадження мультипоточності.

**Спершу вимірюємо час виконання запитів БЕЗ використання мультипоточності.**

Для оцінки продуктивності пошуку книг у системі було виміряно час виконання HTTP-запиту від початку запиту до виведення результатів.

Масив слів складається з **11 слів**:

* Технічних термінів (*python, java, c++, javascript*)
* Загальних термінів (*animal, history, nature*)
* Специфічних термінів (*machine learning, quantum physics*)
* Часто використовуваних слів (*love*)
* Рідкісних слів (*zymurgy*)

Запити для кожного з цих слів по черзі виконуються для різної кількості виведених результатів (*maxResults=5, 10, 20, 40*).

Рахується час, за який для кожного з цих 11 слів по черзі виводиться *maxResults=певному значенню,* і виводиться в консоль.

Маємо відповідну функцію, що виконує це завдання:

def run\_batch\_searches(self):

        search\_words = ["python", "c++", "java", "javascript", "animal",

                        "history", "nature", "machine learning", "quantum physics", "love", "zymurgy"]  # Приклад списку ключових слів

        max\_results\_list = [5, 10, 20, 40]

        for max\_results in max\_results\_list:

            print(f"\n--- Searching with maxResults={max\_results} ---")

            start\_time = time.perf\_counter()  # Починаємо вимір для всіх запитів разом

            for query in search\_words:

                print(f"Searching for '{query}' ...")

                url = f"https://www.googleapis.com/books/v1/volumes?q={query}&maxResults={max\_results}"

                response = requests.get(url)

                if response.status\_code != 200:

                    print(f"Error fetching data for query '{query}'")

                    continue

                data = response.json()

                # Тут можна додати обробку даних, якщо потрібно

            end\_time = time.perf\_counter()  # Кінець виміру для всіх запитів

            elapsed = end\_time - start\_time

            print(f"[Timing] Total time for maxResults={max\_results}: {elapsed:.4f} seconds")

**Результати послідовного виконання запитів**

| **maxResults** | **Загальний час виконання (сек)** |
| --- | --- |
| 5 | 6.0494 |
| 10 | 6.2482 |
| 20 | 6.3079 |
| 40 | 7.3284 |

**Тепер вимірюємо час виконання запитів ІЗ використанням мультипоточності.**

Для впровадження мультипоточності використовуємо пул потоків (ThreadPoolExecutor), де кожен потік одночасно робить свій запит.

Переписуємо нашу функцію run\_batch\_searches() так, щоб вона для кожного значення max\_results (5, 10, 20, 40) запускала пул з 5 потоків (оптимальне значення). Тоді кожен потік паралельно бере по 1 ключовому слову і виконує HTTP-запит до API Google Books.

Таким чином, одночасно працюють до 5 потоків, і загалом всі 11 запитів виконуються швидше, ніж послідовно (один за одним).

Коли один потік закінчує свій запит, пул автоматично запускає наступний запит з черги, поки всі запити не будуть виконані.

Після того, як усі запити для поточного max\_results завершилися, вимірюється загальний час, який зайняло їх паралельне виконання, і виводиться в консоль.

Ось так це виглядає в коді:

def run\_batch\_searches(self):

        search\_words = ["python", "c++", "java", "javascript", "animal",

                        "history", "nature", "machine learning", "quantum physics", "love", "zymurgy"]

        max\_results\_list = [5, 10, 20, 40]

        for max\_results in max\_results\_list:

            print(f"\n--- Searching with maxResults={max\_results} ---")

            start\_time = time.perf\_counter()

            # Функція для одного запиту

            def fetch\_data(query):

                print(f"Searching for '{query}' ...")

                url = f"https://www.googleapis.com/books/v1/volumes?q={query}&maxResults={max\_results}"

                response = requests.get(url)

                if response.status\_code != 200:

                    print(f"Error fetching data for query '{query}'")

                    return None

                return response.json()

            # Виконуємо всі запити паралельно

            with ThreadPoolExecutor(max\_workers=5) as executor:

                # Створюємо майбутні об'єкти (future) для кожного запиту

                futures = {executor.submit(fetch\_data, query): query for query in search\_words}

                for future in as\_completed(futures):

                    query = futures[future]

                    try:

                        data = future.result()

                    except Exception as exc:

                        print(f"Query '{query}' generated an exception: {exc}")

            end\_time = time.perf\_counter()

            elapsed = end\_time - start\_time

            print(f"[Timing] Total time for maxResults={max\_results}: {elapsed:.4f} seconds")

**Результати паралельного виконання запитів**

| **maxResults** | **Загальний час виконання (сек)** |
| --- | --- |
| 5 | 1.8868 |
| 10 | 1.4115 |
| 20 | 1.6902 |
| 40 | 1.7544 |

**Аналізуємо обидві таблиці**

**Паралельне виконання** дає значне прискорення у порівнянні з послідовним.

Час виконання у паралельному режимі — близько 1.4–1.9 секунд, тоді як послідовно — від 6 до понад 7 секунд.

**Це показує, що мультипоточність дозволяє істотно зменшити загальний час очікування.**

**Можемо ще перевірити результати для іншої кількості потоків, але тих самих даних, щоб переконатися, що обрана кількість (5) є оптимальною.**

| **max\_workers** | **maxResults=5** | **maxResults=10** | **maxResults=20** | **maxResults=40** |
| --- | --- | --- | --- | --- |
| 2 | 2.9917 | 3.9882 | 3.6484 | 3.9403 |
| 10 | 1.5742 | 1.0788 | 1.1166 | 1.7928 |
| 20 | 0.9618 | 1.1382 | 0.6060 | 1.3544 |

**max\_workers=2**:

Надто мала кількість потоків. Багато запитів очікують у черзі.

Система не встигає обробляти запити паралельно загальний час високий. Хоч і не такий, як при послідовному виконанні запитів, але порівняно з більшою кількістю потоків - так.

**max\_workers=20**:

Найкращі результати для maxResults=5 і 20, але вже **не стабільно**:

Для maxResults=10 — час вищий, ніж з max\_workers=10.

* для **max\_workers=10** і maxResults=10 час = **1.0788 секунд**
* для **max\_workers=20** і maxResults=10 час = **1.1382 секунд**

Тобто, коли ти **збільшуєш кількість потоків із 10 до 20**, **час обробки збільшується.**

Це вказує на **перевантаження або конкуренцію за ресурси**: надто багато потоків можуть створити затори (особливо на мережевих або системних рівнях).

**max\_workers=10**:

Найбільш **стабільний** варіант з цих:

Час виконання **не коливається сильно** для різних обсягів даних.

Найкращий час для maxResults=10, і гарний баланс у всіх інших випадках.

Результати тут майже такі самі, як і при кількості потоків 5.

**Вочевидь, саме 5-10 потоків достатньо, щоб запити оброблялися майже одночасно, без перевантаження ресурсів.**