Паралельні обчислення в Python: прискорення вашого коду

softserve

ЩО ТАКЕ ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ?

•Паралельні обчислення - це метод виконання обчислень, при якому багато операцій виконуються одночасно. Це досягається шляхом розбиття великої задачі на менші, незалежні підзадачі, які можуть виконуватися одночасно на різних процесорах або ядрах процесора.

•**Аналогія**: Уявіть собі приготування великої вечері. Якщо готує одна людина, це займе багато часу. Але якщо кілька людей готують одночасно, кожен свою страву, вечеря буде готова набагато швидше.

•**Переваги:**

- •**Прискорення виконання коду:** Завдяки одночасному виконанню завдань, загальний час виконання програми значно скорочується.
- •**Ефективне використання ресурсів процесора:** Паралельні обчислення дозволяють використовувати всі доступні ядра процесора, що підвищує продуктивність.

•**Недоліки:**

- •**Складність реалізації:** Паралельне програмування вимагає уважного планування та синхронізації, щоб уникнути помилок.
- •**Ризик виникнення "стану гонитви" (race conditions):** Це ситуація, коли результат виконання програми залежить від порядку виконання потоків або процесів, що може призвести до непередбачуваних помилок.

МОДУЛІ ДЛЯ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

- **threading:**
  - Цей модуль дозволяє створювати потоки (threads) в рамках одного процесу.
  - Потоки ділять між собою пам'ять процесу, що робить обмін даними між ними відносно простим.
  - Однак, через обмеження **GIL** (Global Interpreter Lock), потоки не можуть виконуватися паралельно на кількох ядрах процесора для завдань, які вимагають багато обчислень.
  - **threading** найкраще підходить для завдань, де більшу частину часу займає очікування введення-виводу (наприклад, мережеві запити).
- **multiprocessing:**
  - Цей модуль дозволяє створювати окремі процеси, кожен з яких має власну пам'ять.
  - Процеси можуть виконуватися паралельно на кількох ядрах процесора, що робить **multiprocessing** ідеальним для завдань, які вимагають багато обчислень.
  - Обмін даними між процесами складніший, ніж між потоками, і вимагає використання міжпроцесової взаємодії (IPC).
- **asyncio:**
  - Цей модуль дозволяє писати асинхронний код, який виконується в одному потоці.
  - Асинхронне програмування дозволяє обробляти багато операцій введення-виводу одночасно, не блокуючи виконання програми.
  - **asyncio** найкраще підходить для мережевих додатків та інших завдань, де багато часу витрачається на очікування введення-виводу.

THREADING: ПРИКЛАДИ

```python
import threading
import time
from random import randint


def process_data():
    print("Початок обробки даних...")
    seconds = randint(1, 5)
    print(f"Обробка даних і триватиме {seconds} секунд")
    time.sleep(5)  # Імітація тривалої операції
    print("Обробка даних завершена.")


print("Основний потік: запуск фонової обробки.")
background_thread = threading.Thread(target=process_data)
background_thread.start()

print("Основний потік: продовжує роботу.")
background_thread.join()
print("Основний потік: фонова обробка завершена.")
```

```python
import threading
import time

import requests

urls = ["http://example.com", "http://google.com", "http://python.org"]


def download_url(url):
    print(f"Завантаження {url}")
    response = requests.get(url)
    print(f"Downloaded {url}: {len(response.content)} bytes")


start_time = time.time()
threads = []
for url in urls:
    thread = threading.Thread(target=download_url, args=(url,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

end_time = time.time()
print(f"Загальний час виконання: {end_time - start_time} секунд")
```

softserve

MULTIPROCESSING: ПРИКЛАДИ

```python
import multiprocessing
import time
from random import randint
      You, 10 minutes ago • added examples

def process_chunk(data_chunk):
    # Імітація обробки даних
    seconds = randint(1, 5)
    print(f"Обробка даних: {data_chunk} і триватиме {seconds} секунд")
    time.sleep(seconds)
    print(f"Обробка даних завершена: {data_chunk}")
    return sum(data_chunk)


if __name__ == "__main__":
    data = list(range(100))
    chunk_size = 20
    data_chunks = [data[i : i + chunk_size] for i in range(0, len(data), chunk_size)]

    with multiprocessing.Pool() as pool:
        results = pool.map(process_chunk, data_chunks)

    print(f"Результат обробки: {sum(results)}")
```

```python
import multiprocessing
      You, 11 minutes ago • added examples

def factorial(n):
    print(f"Обчислення факторіалу для числа {n}...")
    result = 1
    for i in range(1, n + 1):
        result *= i
    print(f"Обчислення факторіалу для числа {n} завершено.")
    return result


if __name__ == "__main__":
    numbers = [10, 12, 14, 16, 18, 20]

    with multiprocessing.Pool() as pool:
        results = pool.map(factorial, numbers)

    print(f"Факторіали: {results}")
```

softserve

ASYNCIO: ПРИКЛАДИ

```python
import asyncio

import aiohttp
    You, 11 minutes ago • added examples

async def fetch_data(url):
    print(f"Fetching data from {url}")
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            try:
                return await response.text()
            finally:
                print(f"Finished fetching data from {url}")



async def main():
    urls = [
        "https://jsonplaceholder.typicode.com/todos/1",
        "https://jsonplaceholder.typicode.com/todos/2",
        "https://jsonplaceholder.typicode.com/todos/3",
    ]
    tasks = [fetch_data(url) for url in urls]
    results = await asyncio.gather(*tasks)
    for result in results:
        print(result[:100])


if __name__ == "__main__":
    asyncio.run(main())
```

```python
import asyncio
from random import randint


async def read_file(filename):
    seconds = randint(1, 5)
    print(f"Reading {filename} and Sleeping for {seconds} seconds")
    await asyncio.sleep(seconds)
    with open(filename, "r", encoding="utf-8") as f:
        r = f.read()

    print(f"Finished reading {filename}")
    return r

async def main():
    filenames = ["in\\file1.txt", "in\\file2.txt", "in\\file3.txt"]
    tasks = [read_file(filename) for filename in filenames]
    results = await asyncio.gather(*tasks)
    for result in results:
        print(result)


if __name__ == "__main__":
    asyncio.run(main())
```

softserve

ПОРІВНЯННЯ МОДУЛІВ

Характеристика	threading	multiprocessing	asyncio
Тип паралелізму	**Потоки (Threads)** Виконуються в межах одного процесу, ділячи його ресурси.	**Процеси (Processes)** Кожен процес має власну пам'ять і ресурси, виконуються незалежно.	**Асинхронність (Asynchronous)** Виконує кілька завдань в одному потоці, перемикаючись між ними.
Обмеження	**GIL (Global Interpreter Lock)** Дозволяє лише одному потоку Python виконуватися в один момент часу, обмежуючи паралелізм для CPU-bound завдань.	**Накладні витрати на процеси** Створення та керування процесами вимагає більше ресурсів, ніж потоки.	**Вимагає асинхронного коду** Код повинен бути написаний з використанням async та await, що може бути складним для розуміння.
Використання	**Операції введення-виводу (I/O)**: Ефективний для завдань, де багато часу витрачається на очікування введення-виводу (наприклад, мережеві запити).	**Обчислювальні завдання (CPU-bound)**: Ідеально підходить для завдань, які вимагають багато обчислень і можуть використовувати кілька ядер.	**Мережеві операції, I/O**: Оптимізований для завдань, де потрібно обробляти багато одночасних операцій введення-виводу.
Швидкість	**Швидко для I/O, обмежено GIL**: Швидко для завдань, які не вимагають багато обчислень, але обмежений GIL для CPU-bound завдань.	**Швидко для CPU-bound, накладні витрати на створення процесів**: Швидко для обчислювальних завдань, але має накладні витрати на створення процесів.	**Швидко для I/O, один потік**: Дуже швидко для завдань, де багато часу витрачається на очікування введення-виводу, але виконується в одному потоці.
Складність	**Відносно просто, але потрібно враховувати GIL**: Легко реалізувати, але потрібно розуміти обмеження GIL і уникати race conditions.	**Складніше, ніж threading**: Вимагає більше коду для керування процесами та обміну даними.	**Складніше, ніж threading, вимагає асинхронного мислення**: Вимагає розуміння асинхронного програмування та використання async/await.
Обмін даними	**Спільна пам'ять, ризик race conditions**: Потоки ділять спільну пам'ять, що може призвести до race conditions, якщо не використовувати синхронізацію.	**IPC (Inter-Process Communication), більше накладних витрат**: Процеси мають власну пам'ять, тому обмін даними вимагає IPC, що має накладні витрати.	**Події, колбеки, асинхронні черги**: Використовує події, колбеки та асинхронні черги для обміну даними, що може бути складним для розуміння.
Пам'ять	**Спільна пам'ять, менше накладних витрат**: Потоки ділять спільну пам'ять, тому мають менше накладних витрат.	**Окрема пам'ять для кожного процесу, більше накладних витрат**: Кожен процес має власну пам'ять, тому має більше накладних витрат.	**Один потік, менше накладних витрат**: Виконується в одному потоці, тому має менше накладних витрат.
Приклади використання	**Завантаження веб-сторінок, фонові завдання**: Завантаження кількох веб-сторінок одночасно, обробка даних у фоновому режимі.	**Обробка великих даних, математичні обчислення**: Обробка великих обсягів даних, виконання складних математичних обчислень.	**Мережеві сервери, веб-скрейпінг, чат-боти**: Обробка мережевих запитів, веб-скрейпінг, створення чат-ботів.

ВИБІР ПРАВИЛЬНОГО ІНСТРУМЕНТУ

- Тип завдання:
  - Якщо завдання пов'язане з операціями введення-виводу (I/O), такими як мережеві запити, читання/запис файлів, то найкраще підходять модулі **threading** або **asyncio**.
  - Якщо завдання вимагає багато обчислень (CPU-bound), наприклад, обробка великих обсягів даних, математичні розрахунки, то слід використовувати модуль **multiprocessing**.
- Обмеження модулів:
  - Необхідно враховувати обмеження кожного модуля. Наприклад, **threading** обмежений GIL, що може знизити продуктивність для CPU-bound завдань.
  - **multiprocessing** створює окремі процеси, що вимагає більше ресурсів і ускладнює обмін даними.
  - **asyncio** вимагає асинхронного програмування, що може бути складним для розуміння.
- Складність реалізації:
  - Необхідно оцінити складність реалізації паралельного коду з використанням кожного модуля.
  - **threading** відносно простий у використанні, але вимагає уваги до синхронізації.
  - **multiprocessing** складніший через необхідність обміну даними між процесами.
  - **asyncio** вимагає розуміння асинхронного програмування.

ВИСНОВКИ

- Паралельні обчислення прискорюють код:
  - Паралельні обчислення є потужним інструментом для підвищення продуктивності програм.

- Python надає різні інструменти:
  - Python надає три основні модулі для паралельних обчислень: **threading**, **multiprocessing** і **asyncio**.

- Важливість вибору правильного інструменту:
  - Вибір модуля залежить від конкретного завдання і необхідно враховувати його особливості.

Q&A

softserve