Київський національний університет імені Т. Шевченка

Факультет комп’ютерних наук та кібернетики

Звіт

з дисципліни “Розподілене та паралельне програмування”

за темою: “Паралельне множення матриць (блочний алгоритм)”

Виконав

студент 4 курсу, групи ТК-41

спеціальності 122 “Комп'ютерні науки”

Панасюк Кирил

Керівник Деревяченко О.В.

Київ - 2024

Зміст

1. Мета роботи

Метою лабораторної роботи є ознайомлення з принципами реалізації паралельних обчислень на прикладі множення квадратних матриць з використанням блочного алгоритму. У ході виконання завдання реалізовано три варіанти множення:

- послідовний (класичний) метод;
- паралельний метод з використанням технології **OpenMP** (у нашому випадку через модуль multiprocessing);
- паралельний метод з використанням бібліотеки **MPI** (mpi4py).

Завданням є порівняння продуктивності кожного з підходів шляхом запуску програми з однаковими вхідними даними, вимірювання часу виконання та побудова порівняльної таблиці. Це дозволяє на практиці оцінити переваги паралельної обробки даних та ефективність обраних технологій у задачах великої обчислювальної складності.

2. Теоретичні відомості

2.1. Поняття паралельного програмування

Паралельне програмування — це підхід до розробки програмного забезпечення, при якому обчислення розподіляються між кількома процесами або потоками, які можуть виконуватися одночасно. Такий підхід дозволяє скорочувати час виконання програм, особливо в задачах з великою обчислювальною складністю.

У контексті сучасних комп’ютерних систем, що мають багатоядерні процесори або доступ до кластерів, паралельне програмування є ключовим інструментом для підвищення продуктивності.

2.2. Переваги паралельних обчислень

- **Підвищення швидкості обробки даних** — завдяки розподілу задач між кількома ядрами або вузлами.

- **Краще використання ресурсів системи** — забезпечується навантаженням усіх ядер.

- **Масштабованість** — можливість обробки великих обсягів даних за допомогою кластерних систем.

- **Ефективність у реальному часі** — підвищена реакція систем у задачах з високою вимогливістю до часу виконання.

2.3. Технології MPI та OpenMP

MPI (Message Passing Interface)

MPI — це стандарт для обміну повідомленнями між процесами в паралельних обчисленнях, які зазвичай виконуються на різних вузлах кластера або машин. У Python реалізується через бібліотеку mpi4py.
Кожен процес має власну пам’ять, обмін даними відбувається через операції send, recv, bcast, scatter, gather тощо.

OpenMP (Open Multi-Processing)

OpenMP — технологія для організації паралелізму за допомогою потоків у межах одного процесу. У Python її аналогом є використання модуля multiprocessing, що дозволяє запускати функції в окремих процесах на різних ядрах CPU.

2.4. Суть блочного алгоритму множення матриць

Блочний алгоритм множення матриць передбачає поділ великих матриць на менші блоки (субматриці), які можуть бути незалежно оброблені окремими процесами або потоками. Це дозволяє:

- зменшити час доступу до пам’яті;

- знизити кількість обмінів між процесами (в MPI);

- ефективніше використовувати кеш-пам’ять CPU.

У реалізаціях цієї лабораторної роботи використовувався умовний розподіл матриць по рядках між процесами. Кожен процес обчислює свою частину результатної матриці, після чого дані об’єднуються (gather).

3. Опис реалізації

3.1. Послідовна реалізація

У цьому варіанті використовується вбудована функція np.dot() з бібліотеки NumPy, яка виконує множення двох квадратних матриць стандартним алгоритмом. Обчислення виконуються в одному потоці — кожен елемент результатної матриці розраховується по черзі, що забезпечує простоту реалізації, але низьку ефективність при великому обсязі даних.

Фрагмент коду:

```
def sequential_multiply(A, B):
    return np.dot(A, B)
```

3.2. Паралельна реалізація з використанням OpenMP (модуль multiprocessing)

Оскільки OpenMP безпосередньо не підтримується в Python, аналогічний підхід реалізовано за допомогою модуля multiprocessing, який створює пул процесів для обробки рядків матриці.

Кожен процес обчислює добуток одного рядка першої матриці на всю другу матрицю. Це дозволяє значно пришвидшити обчислення при наявності кількох ядер CPU.

Фрагмент коду:

```
def parallel_worker(args):
    A_row, B = args
    return np.dot(A_row, B)

def openmp_multiply(A, B):
    with multiprocessing.Pool() as pool:
        result = pool.map(parallel_worker, [(row, B) for row in A])
```

```
    return np.array(result)
```

3.3. Паралельна реалізація з використанням MPI (бібліотека mpi4py)

У реалізації з MPI кожен процес отримує частину рядків матриці A через Scatter, а повну матрицю B через Bcast. Потім виконується множення виділеної частини та збір результатів з усіх процесів через Gather.

Такий підхід є ідеальним для кластерних середовищ, де кожен процес може виконуватися на окремому вузлі.

Фрагмент коду:

```
def mpi_multiply(A, B, n):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    rows_per_proc = n // size

    local_A = np.zeros((rows_per_proc, n), dtype='i')
    B_global = np.zeros((n, n), dtype='i')
    C = np.zeros((n, n), dtype='i') if rank == 0 else None

    if rank == 0:
        comm.Bcast(B, root=0)
        comm.Scatter([A, MPI.INT], [local_A, MPI.INT], root=0)
    else:
        comm.Bcast(B_global, root=0)
        B = B_global
        comm.Scatter([None, MPI.INT], [local_A, MPI.INT], root=0)

    local_C = np.dot(local_A, B)
    comm.Gather(local_C, C, root=0)
    return C
```

4. Результати експериментів

Для оцінки ефективності реалізованих алгоритмів було проведено серію тестів (по 3 заміри для кожного методу) на матрицях розміру **4×4**. Нижче представлено результати вимірювання часу виконання для кожного з варіантів:

Таблиця 1. Час виконання для кожного способу множення матриць

№ тесту	Послідовний (сек.)	OpenMP (сек.)	MPI (сек.)
1	0.0000	1.1224	0.0025
2	0.0000	0.9601	0.0002
3	0.0000	0.9380	0.0002

Середній час виконання

Для об'єктивного порівняння було обчислено середнє арифметичне часу виконання для кожного методу:

- **Послідовний метод**: (0.0000 + 0.0000 + 0.0000) / 3 = 0.0000 сек
- **OpenMP (multiprocessing)**: (1.1224 + 0.9601 + 0.9380) / 3 ≈ 1.0068 сек
- **MPI**: (0.0025 + 0.0002 + 0.0002) / 3 ≈ 0.00097 сек

5. Висновки

У ході виконання лабораторної роботи було реалізовано три способи множення квадратних матриць: послідовний, паралельний із використанням OpenMP (через multiprocessing) та паралельний із використанням MPI (mpi4py). Після проведення експериментальних замірів часу виконання було отримано наступні результати:

- **Послідовний метод** продемонстрував найменший час виконання (0.0000 сек), що пов’язано з малим розміром матриць (4×4) та відсутністю витрат на створення процесів чи передачу даних.

- **OpenMP (multiprocessing)** показав найбільший час виконання (≈1 секунда), оскільки витрати на створення пулу процесів і міжпроцесну комунікацію перевищують вигоди від паралелізму при малих задачах.

- **MPI** продемонстрував стабільно малий час виконання (≈0.001 сек), оскільки розподіл даних по процесах був ефективним, і сама задача мала низьке навантаження.

На мою думку, ця робота ще раз доводить, що паралельні алгоритми не завжди є ефективними при невеликих обсягах даних. У випадку з малими матрицями паралельні технології можуть навіть знижувати продуктивність через накладні витрати.

Додаток А

```
import numpy as np
import time
import multiprocessing
from mpi4py import MPI

# --- Генерація випадкових матриць ---
def generate_matrices(n):
    A = np.random.randint(0, 10, (n, n))
    B = np.random.randint(0, 10, (n, n))
    return A, B

# --- Послідовне множення ---
def sequential_multiply(A, B):
    return np.dot(A, B)

# --- Паралельне множення (OpenMP-подібне через multiprocessing)
---
def parallel_worker(args):
    A_row, B = args
    return np.dot(A_row, B)

def openmp_multiply(A, B):
    with multiprocessing.Pool() as pool:
        result = pool.map(parallel_worker, [(row, B) for row in A])
    return np.array(result)

# --- Паралельне множення (MPI) ---
def mpi_multiply(A, B, n):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    rows_per_proc = n // size
    local_A = np.zeros((rows_per_proc, n), dtype='i')
    B_global = np.zeros((n, n), dtype='i')
    C = np.zeros((n, n), dtype='i') if rank == 0 else None

    if rank == 0:
        comm.Bcast(B, root=0)
        comm.Scatter([A, MPI.INT], [local_A, MPI.INT], root=0)
    else:
        comm.Bcast(B_global, root=0)
        B = B_global
        comm.Scatter([None, MPI.INT], [local_A, MPI.INT], root=0)

    local_C = np.dot(local_A, B)
```

```
    comm.Gather(local_C, C, root=0)
    return C

def main():
    n = 4  # Розмір матриці
    print("Оберіть реалізацію:")
    print("1 - Послідовна")
    print("2 - Паралельна (OpenMP / multiprocessing)")
    print("3 - Паралельна (MPI)")
    choice = input("Ваш вибір: ")

    if choice not in ['1', '2', '3']:
        print("Невірний вибір!")
        return

    if choice == '3':
        comm = MPI.COMM_WORLD
        rank = comm.Get_rank()
        if rank == 0:
            A, B = generate_matrices(n)
        else:
            A = None
            B = None
        start = MPI.Wtime()
        C = mpi_multiply(A, B, n)
        end = MPI.Wtime()
        if rank == 0:
            print(f"[MPI] Час виконання: {end - start:.4f} секунд")
            print("\nМатриця A:\n", A)
            print("\nМатриця B:\n", B)
            print("\nРезультат множення (C = A * B):\n", C)
    else:
        A, B = generate_matrices(n)
        start = time.time()
        if choice == '1':
            C = sequential_multiply(A, B)
        elif choice == '2':
            C = openmp_multiply(A, B)
        end = time.time()
        method = "Послідовна" if choice == '1' else "OpenMP
(multiprocessing)"
        print(f"[{method}] Час виконання: {end - start:.4f}
секунд")
        print("\nМатриця A:\n", A)
        print("\nМатриця B:\n", B)
        print("\nРезультат множення (C = A * B):\n", C)
```

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